MuPAD
O software de álgebra livre e aberto

Prefácio
Este livro explica o uso básico do pacote de software chamado MuPAD e dá um insight dentro do poder do sistema. MuPAD é também chamado de sistema de álgebra computacional, que é desenvolvido principalmente pela Sciface Software e o grupo de pesquisa MuPAD da universidade de Paderborn na Alemanha.

Esta introdução é endereçada a matemáticos, engenheiros, cientistas da computação, cientistas da natureza e, de modo mais geral, a todos que precisem de computações matemáticas para sua educação ou sua profissão.
Geralmente falando, este livro é endereçado a qualquer um que queira usar o poder de um moderno pacote de álgebra computacional.

Há dois meios de se usar um sistema de álgebra computacional. No primeiro deles, você pode usar o conhecimento matemático incorporado para manipular sistemas interativamente. Por exemplo, você pode computar integrais simbólicas ou gerar e inverter matrizes pelas funções apropriadas. Isto consta a inteligência matemática do sistema e pode implementar algoritmos sofisticados. Do capítulo 2 até 15 discute este meio de se usar o MuPAD.

Na outra mão, com a ajuda da linguagem de programação do MuPAD, você pode facilmente adicionar funcionalidades para o sistema implementando seus próprios algoritmos como procedimentos no MuPAD. Isso é útil para objetivos de aplicações especiais ou então sistemas apropriados de funções existentes. Capítulo 16 até 18 é uma introdução à programação no MuPAD.

Para iniciantes no MuPAD, nós recomendamos começar a ler do capítulo 2, que dá um pequeno guia de sobrevivência do MuPAD. A descrição do sistema de ajuda onlinje na seção 2.1 provavelmente é a mais importante parte deste capítulo. O sistema de ajuda provê informações sobre detalhes das funções do sistema, sua sintaxe, seus parâmetros de controle, etc. Isto é avaliado online durante uma sessão do MuPAD. O capítulo 2 demonstra alguns dos mais importantes sistemas funcionais “em trabalho”. Você vai encontrar futuros detalhes sobre estas funções nas partes mais adiantes do livro ou nas páginas de ajuda. Para um profundo entendimento das estruturas de dados envolvidas, você pode consultar a sessão correspondente no capítulo 4.

Capítulo 3 discute as bibliotecas do MuPAD e seus usos. Elas contém muitas funções e algoritmos para tópicos matemáticos particulares.

Os tipos básicos de dados e o sistema mais importante de funções para seu controle são introduzidos no capítulo 4. Não há necessidade de estudar todos eles com a mesma intensidade. Dependendo do que você queira aplicar você pode sewletivamente ler aquilo que for relevante pra você.

O capítulo 5 explica como o MuPAD iguala objetos; nós recomendamos fortemente a ler este capítulo.

O capítulo 6 até o 11 discutem o uso de alguns sistemas de funções particularmente importantes: substituição, diferenciação, geração de números aleatórios, e comandos gráficos.

Várias participações úteis como a história do mecanismo, rotinas de entrada e saída, ou a definição das preferências do usuário são descritas do capítulo 12 até 14. Preferências podem ser usadas para configurar a interatividade do comportamento do sistema depois que o usuário têm uma certa imaginação externa.

Capítulo 16 até o 18 dá uma introdução para o conceito técnico da linguagem de programação do MuPAD.

MuPAD provê algoritmos que podem controlar uma grande classe de objetos matemáticos e tarefas relacionadas. Se você ler esta introdução, será possível que você encontre noções matemáticas desconhecidas como campos e anéis. Esta introdução não pretende explicar o background matemático de tais objetos. O conhecimento matemático básico é útil mas não obrigatório para entender o texto. Algumas vezes você pode perguntar como o MuPAD usa os algoritmos para resolver um problema em particular. O modo interno de operação do MuPAD não se encontra aqui: nós não pretendemos dar uma introdução geral para álgebra computacional e seus algoritmos. Os leitores interessados podem consultar os text books como, por exemplo [GCL92] ou [GG99].

Este livro dá uma introdução primária para o MuPAD. Objetos um pouco mais abstratos como, por exemplo, extensões de campo, são fáceis de descrever e manipular no MuPAD. Entretanto, tais aspectos avançados do sistema não são discutidos aqui. As aplicações matemáticas que são mencionadas no texto são intencionalmente mantidas em um nível primário. Isto é para manter este texto claro para leitores com pouco conhecimento matemático e para fazer isto aplicável em nível escolar.

Nós não podemos explicar o completo funcionamento do MuPAD nesta introdução. Algumas partes do sistema são mencionados somente levemente. Isto está além do âmbito deste tutorial para seguir dentro dos detalhes completos do poder da linguagem de programação do MuPAD. Você encontra isso no sistema de ajuda do MuPAD reconhecido online durante uma sessão no MuPAD.

Este tutorial refere-se as versões 3.0 e as seguintes. Desde o desenvolvimento do sistema foram feitos avanços contínuos, alguns dos detalhes descritos podem mudar no futuro. Futuras versões vão definitivamente providenciar funcionalidades adicionais através de novos sistemas de funções e pacotes de aplicações. Neste tutorial, nós apresentamos principalmente as ferramentas básicas e seus usos, dos quais irão provavelmente permanecer essencialmente inalterados. Nós tentamos colocar palavras com todas as declarações no texto de tal forma que elas ficassem basicamente válidas para versões futuras do MuPAD.

Paderborn, Março 2004

Índice

Prefácio.....**1**

1 Introdução.....**3**
1.1 Computações numéricas.....3
1.2 Álgebra computacional.....3
1.3 Características de sistemas de álgebra computacionais.....4
1.4 Sistemas existentes.....4
1.5 MuPAD.....4

2 Primeiros passos no MuPAD.....**5**
2.1 Explicações e ajuda.....6
2.2 Computando com números.....7
2.2.1 Computações exatas.....8
2.2.2 Aproximações numéricas.....8
2.2.3 Números complexos.....9
2.3 Computação simbólica.....10
2.3.1 Exemplos introdutórios.....10
2.3.2 Esboço de curva.....14
2.3.3 Teoria elementar dos números.....15

3 As bibliotecas do MuPAD.....**18**
3.1 Informações sobre uma biblioteca particular.....18
3.2 Exportando bibliotecas.....19
3.3 A biblioteca padrão.....20

4 Objetos do MuPAD.....**20**
4.1 Operandos: as funções op e nops.....21
4.2 Números.....22
4.3 Identificadores.....23
4.4 Expressões simbólicas.....25
4.4.1 Operadores.....25
4.4.2 Árvores de expressão.....28
4.4.3 Operandos.....29
4.5 Sequências.....30
4.6 Listas.....32
4.7 Conjuntos.....35
4.8 Tabelas.....36
4.9 Arranjos.....37
4.10 Expressões Booleanas.....39
4.11 Fileiras.....39
4.12 Funções.....40
4.13 Expansões de séries.....41
4.14 Estruturas algébricas: campos, anéis, etc.....43
4.15 Vetores e matrizes.....45
4.15.1 Definições de matrizes e vetores.....45
4.15.2 Computando com matrizes.....47
4.15.3 Métodos especiais para matrizes.....48
4.15.4 As bibliotecas linalg e numeric.....49
4.15.5 Matrizes escassas.....50
4.15.6 Uma aplicação.....51
4.16 Polinômios.....52
4.16.1 Definição de polinômios.....52

4.16.2 Computando com polinômios.....54
4.17 Aritmética de intervalo.....56
4.18 objetos nulos: null ( ), NIL, FAIL, undefinied.....58

5 Avaliação e simplificação.....59
5.1 Identificadores e seus valores.....59
5.2 Completa, incompleta e avaliação forçada.....60
5.3 Simplificação automática.....62

6 Substituição: subs, subsex, e subsop.....63

7 Diferenciação e integração.....65
7.1 Diferenciação.....65
7.2 Integração.....66

8 Resolvendo equações: solve.....67
8.1 Equações polinomiais.....67
8.2 Equações gerais e inequações.....69
8.3 Equações diferenciais.....71
8.4 Equações recorrentes.....72

9 Manipulando expressões.....72
9.1 Transformando expressões.....72
9.2 Simplificando expressões.....76
9.3 Suposições sobre identificadores simbólicos.....78

10 Chance e probabilidade.....81

11 Gráficos.....84

12 A história dos mecanismos.....84
13 Entrada e saída.....85
13.1 Saídas de expressões.....85
13.1.1 Imprimindo expressões na tela.....85
13.1.2 Modificando o formato de saída.....85
13.2 Lendo os arquivos escritos.....86
13.2.1 As funções **write** e **read**.....86
13.2.2 Salvando uma sessão do MuPAD.....87
13.2.3 Lendo dados de um arquivo de texto.....87

14 Utilidades.....87
14.1 Preferências definidas pelo usuário.....87
14.2 Informação nos algoritmos do MuPAD.....89
14.3 Reiniciando uma sessão do MuPAD.....90
14.4 Executando comandos do sistema operacional.....90

15 Tipos de especificadores.....90
15.1 As funções **type** e **testtype**.....90
15.2 Tipo de checagem confortável: a biblioteca **Type**.....91

16 Loops.....92

17 Ramificando: if-then-else e **case.....94**

18 Procedimentos do MuPAD.....96
18.1 Definindo Procedimentos.....96
18.2 O valor retornado de um procedimento.....97
18.3 Retornando chamadas de funções simbólicas.....97
18.4 variáveis locais e globais.....98
18.5 Subprocedimentos.....100
18.6 Extensão de variáveis.....101
18.7 Tipo declaração.....102
18.8 Procedimentos com um número variável de argumentos.....103
18.9 Opções: a tabela de lembrança.....103
18.10 Parâmetros de entrada.....105
18.11 Avaliação dentro de procedimentos.....106
18.12 Funções Ambientes.....106
18.13 Um exemplo de programação: Diferenciação.....109
18.14 Programando exercícios.....110

Capítulo 1 Introdução

Para explicar noção de álgebra computacional, nós comparamos computações algébricas e numéricas. Ambos os gêneros são suportados por um computador, mas há diferenças fundamentais, das quais serão discutidas no que se segue.

1.1 Computações numéricas

Um problema matemático pode ser resolvido aproximadamente por computações numéricas. Os passos da computação operam em números, que são internamente armazenados em representações de pontos flutuantes. Esta representação tem a desvantagem que nenhuma computação ou solução são exatas devido a arredondamento e erros. Em geral, algoritmos numéricos encontram soluções aproximadas tão rápido quanto possível. Frequentemente soluções como essas são o único caminho para controlar um problema matemático computacionalmente, em particular se não há uma fórmula fechada de soluções conhecidas. Mais ainda, soluções aproximadas são úteis se os resultados exatos não são necessários (por exemplo, na visualização).

1.2 Álgebra computacional

Em contraste com computações numéricas, há computações simbólicas na álgebra computacional, elas são definidas como "computações com símbolos representando um assunto matemático". Aqui,um assunto pode ser um número, mas também um polinômio, uma equação, uma expressão ou uma fórmula, uma função, um grupo(conjunto), um anel, ou qualquer outro objeto matemático. Computações simbólicas com números são levados fora do exato. Internamente números são representados como quocientes ou integrais de peso arbitrário (limitado pela quantidade de armazenamento disponível, por exemplo). Esses gêneros de computações são chamados simbólico e algébrico. Dadas as seguintes definições:

1. "Simbólica" enfatiza o que em muitos casos (o último gol do problema matemático resolvido é expresso a resposta em uma fórmula fechada ou encontrando uma aproximação simbólica
2. "Algébrica" significa que computações são levadas além do exato, de acordo com as regras da álgebra, ao invés de usar pontos flutuantes e aproximações aritméticas.

As vezes "manipulação simbólica" ou "manipulação de fórmula" são usados como sinônimos para álgebra computacional, desde que computações operem em símbolos e formulários. Os exemplos são integração simbólica ou diferenciação tais como

$$\int x\,dx = \frac{x^2}{2}, \int_1^4 x\,x\,dx = \frac{15}{2}, \frac{d}{dx}\ln\ln x = \frac{1}{x\ln x}$$

Ou soluções simbólicas de equações. Por exemplo, nós consideramos a equação $x^4+px^2+1=0$ em x com parâmetro p. Sua solução jogada é

$$\left\{\pm\frac{\sqrt{2}\sqrt{-p-\sqrt{p^2-4}}}{2}, \pm\frac{\sqrt{2}\sqrt{-p+\sqrt{p^2-4}}}{2}\right\}.$$

A computação simbólica de uma solução exata normalmente requer mais tempo processando e mais espaço que computação numérica de uma solução aproximada. Entretanto, uma solução simbólica é exata, mais geralmente, e muitas vezes prove mais informação sobre o problema e sua solução. A fórmula acima expressa a solução da equação em termos do parâmetro p. Isso mostra como a solução depende funcionalmente em p. Esta informação pode ser usada por exemplo, para examinar com uma sensível solução como se comporta quando o parâmetro muda.

Combinações de métodos simbólicos e numéricos são úteis para aplicações especiais. Por exemplo, há algoritmos na álgebra computacional que beneficia ao eficiente hardware de pontos flutuantes aritméticos. Na outra mão, isso pode ser útil para simplificar um problema de análise numérica simbolicamente antes de aplicar a atual aproximação de algoritmo.

1.3. Características de sistemas de álgebra computacionais

A maioria dos atuais sistemas de álgebra computacionais podem ser usados interativamente. O usuário entra com alguma fórmula e comando, e o sistema avalia isto. Aí retorna uma resposta, que pode ser manipulada futuramente se necessário.

Em adição à computações simbólicas, a maioria dos pacotes de álgebra computacional pode aproximar soluções numericamente também. O usuário pode jogar a precisão a um número de dígitos desejado. No MuPAD, a variável global DIGITS manipula isso. Por exemplo, se você entrar com o simples comando DIGITS:=100, então o MuPAD apresenta todos os pontos flutuantes com a precisão de 100 dígitos decimais. Certamente, tais computações precisam de mais tempo e mais espaço que o uso dos pontos flutuantes do hardware aritmético.
Mais ainda, sistemas modernos de álgebra computacional proveem uma poderosa linguagem de programa e ferramentas para visualização e animação de dados matemáticos. Também, muitos sistemas podem produzir planos de documentos, (conhecidos como cadernos ou folhas de trabalho). MuPAD têm um conceito parecido com caderno, mas nós não descrevemos isso neste tutorial. O gol deste livro é dar uma introdução ao poder matemático do MuPAD.

1.4 Sistemas existentes

Há muitos sistemas diferentes de álgebra computacional. Alguns deles são distribuídos comercialmente, enquanto outros podem ser obtidos de graça.

Sistemas de propósitos especiais podem controlar problemas matemáticos particulares. Por exemplo, o sistema Schoonship é designado para problemas em energias físicas altas. DeLiA para equações diferenciais. PARI para aplicações em teoria dos números, e GAP para problemas em teoria de grupo.

Então eles são ou propósito geral de sistemas de álgebra computacionais. Derive foi desenvolvido desde 1980 e foi designado especialmente para minicomputadores. Math View (antigo Theorist), desenvolvido desde 1990, tem uma interface elaborada de números, mas somente compatível com um pouco de conhecimento matemático. Os sistemas Macsyma e Reduce ambos começaram em 1965 e são programados em LISP, assim como Maxima, uma variação de Macsyma começado em 1998. Sistemas poderosos como Mathematica e Maple, desenvolvidos desde o começo dos anos 80, são programados em C. Mathematica foi o primeiro sistema com uma interface de usuário amigável. Em contraste com os sistemas acima, Axiom teve uma linguagem de tipo completa, e computações tomaram lugar em contextos matemáticos específicos. MuPAD é o mais jovem desses sistemas gerais. Ele foi desenvolvido na universidade de Paderborn desde 1990, e tenta combinar os méritos desses predecessores com conceitos modernos deste próprio.

1.5 MuPAD

Em adição a começar as propriedades de sistemas algébricos computacionais, MuPAD têm as seguintes características, que não serão discutidas em detalhes neste livro.

- MuPAD provê construções de linguagem para assunto orientando a programação. Você endereçar objeto – orientado pode definir seus próprios tipos de dados. A maioria de todas as operações existentes e funções pode ser sobrecarregada.

- MuPAD oferece um depurador de código de fonte interativo.

- Programas escritos em C ou C++ podem ser adicionados ao kernel pelo conceito de módulo dinâmico do MuPAD.

O coração do MuPAD é o seu kernel que é simplesmente em C e em parte em C++. Isso cumpre os seguintes componentes principais:

- O parser lê a entrada do sistema e apresenta uma sintaxe avaliada. Se nenhum erro for encontrado, ele converte a entrada para os tipos de dados do MuPAD.

- O processo avaliador processa e simplifica a entrada. Este modo de operação será discutido mais pra frente.

- A memória gerente é responsável pelo eficiente espaço de objetos do MuPAD.

- Alguns algoritmos frequentemente usados tais como, por exemplo, funções aritméticas são implementados como funções do kernel em C.

Parser e avaliador definem a linguagem do MuPAD como descrito neste livro. As bibliotecas do MuPAD, as quais contêm a maioria do conhecimento matemático do sistema, são implementadas nesta linguagem.

Em adição, MuPAD oferece uma interface de usuário confortável por gerar cadernos ou gráficos ou para depurar programas escritos na linguagem do MuPAD. O sistema de ajuda do MuPAD tem uma funcionalidade hipertextual. Você pode navegar dentro de documentos e executar exemplos por um clique do mouse. Figura 1.1 mostra os principais componentes do sistema MuPAD.

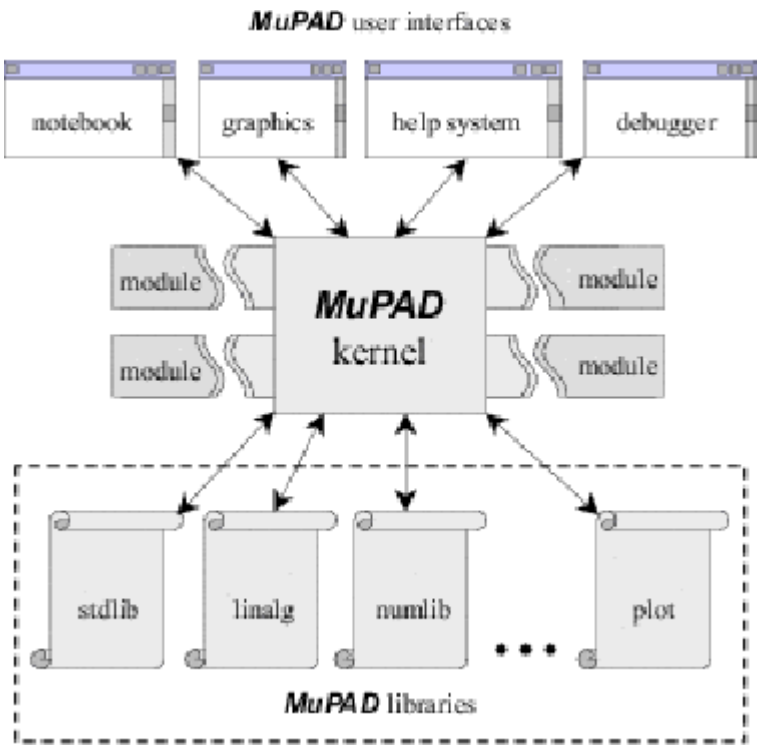


Figura 1.1: principais componentes do MuPAD

Capítulo 2

Primeiros passos no MuPAD

Sistemas de álgebra computacionais como MuPAD são oferecidos para serem usados interativamente. Por exemplo, você pode entrar com uma instrução para multiplicar dois números e esperar até o MuPAD computar o resultado e imprimir ele na tela.

Depois que você chamar o programa MuPAD, uma sessão é lançada. Você encontra a informação de como começar no programa MuPAD nas instruções de instalação para seu sistema operacional. MuPAD provê um sistema de ajuda que você pode consultar durante uma sessão para encontrar detalhes sobre funções do sistema, sua sintaxe, seus parâmetros, etc. A sessão seguinte apresenta uma introdução ao sistema de ajuda do MuPAD. Requisitar uma página de ajuda é provavelmente o comando mais frequentemente usado por iniciantes. A sessão que vêm depois desta é sobre usar o MuPAD como uma “calculadora de bolso inteligente”: calculando com números. Esta é a mais fácil e a mais intuitiva parte desse tutorial. Depois nós introduzimos alguns sistemas de funções para computações simbólicas. A sessão correspondente é escrita informalmente e dá um primeiro insight dentro das características simbólicas do sistema.

Depois de iniciar o programa, você pode entrar com comamndo na linguagem do MuPAD. O sistema espera sua entrada quando o prompt do MuPAD aparece na tela com o sinal ⌐, enquanto é >> em plataformas UNIX . Nós usaremos o prompt UNIX em

todos os exemplos através deste livro. Se você pressionar <RETURN> em Windows ou UNIX, ele completa sua entrada e o MuPAD avalia o comando que você entrou. Segurando <SHIFT> enquanto pressiona <RETURN> apenas provoca um alimento de linha e ainda leva MuPAD ao modo de entrada. No Macintosh, <SHIFT> + <RETURN> ou <ENTER> executa um comando e <RETURN> apenas provoca uma linha de alimentação. Para todas as versões GUI, você pode trocar os papéis de <RETURN> e <SHIFT>+<RETURN> escolhendo “Opções” no menu “View” e então clicando em “Enter only.” Se você entrar:

```
>> sin(3.141)
```

E então pressionar <RETURN> (ou <ENTER>, respectivamente), o resultado

0.0005926535551

É impresso na tela. O sistema avalia a função usual seno no ponto 3.141 e retorna um ponto flutuante aproximado do valor, similar a saída de uma calculadora de bolso.

Se você terminar seu comando com uma coluna, então o MuPAD executa o comando sem imprimir este resultado na tela. Isso habilita você para suprimir a saída de resultados intermediários irrelevantes. Você pode entrar com mais de um comando em uma linha. Dois comandos subsequentes têm que estar separados por uma semicoluna ou uma coluna, se o resultado do primeiro comando é para ser suprimido ou não, respectivamente:

```
>> diff(sin(x^2), x); int(%, x)
```

$$2\,x\,\cos\left(x^2\right)$$

$$\sin\left(x^2\right)$$

Aqui x^2 denota o quadrado de x, e a função diff e int representa as operações “diferenciais” e “integrais” (Capítulo 7). O caractere % retorna a expressão anterior (no exemplo, esta é derivada de sin (x^2)). O conceito estando por baixo de % é discutido no Capítulo 12.

No seguinte exemplo, a saída do primeiro comando é extra numericamente por uma coluna, e apenas o resultado do segundo comando aparece na tela.

```
>> equations := {x + y = 1, x - y = 1}:
>> solve(equations)
```

$$\{[x = 1, y = 0]\}$$

No exemplo anterior, um jogo de duas equações é assinalado pela identidade equações. O comando solve (equações) computa a solução. Capítulo 8 discute o solve em mais detalhes.

Na terminal versão do MuPAD (em sistemas UNIX), você pode terminar a atual sessão do MuPAD entrando com a palavra chave quit:

```
>> quit
```

Versões do MuPAD com uma interface gráfica podem ser finalizadas com o menu correspondente registrado no GUI

2.1 Explicações e ajuda

Se você não sabe a sintaxe correta de um comando do MuPAD, você pode obter esta informação diretamente do sistema de ajuda online. Para muitas das rotinas do MuPAD, a função info retorna uma breve explicação:

```
>> info(solve)
   solve -- solve equations and inequalities [try ?solve\
    for options]

>> info(ln)
   ln -- the natural logarithm
```

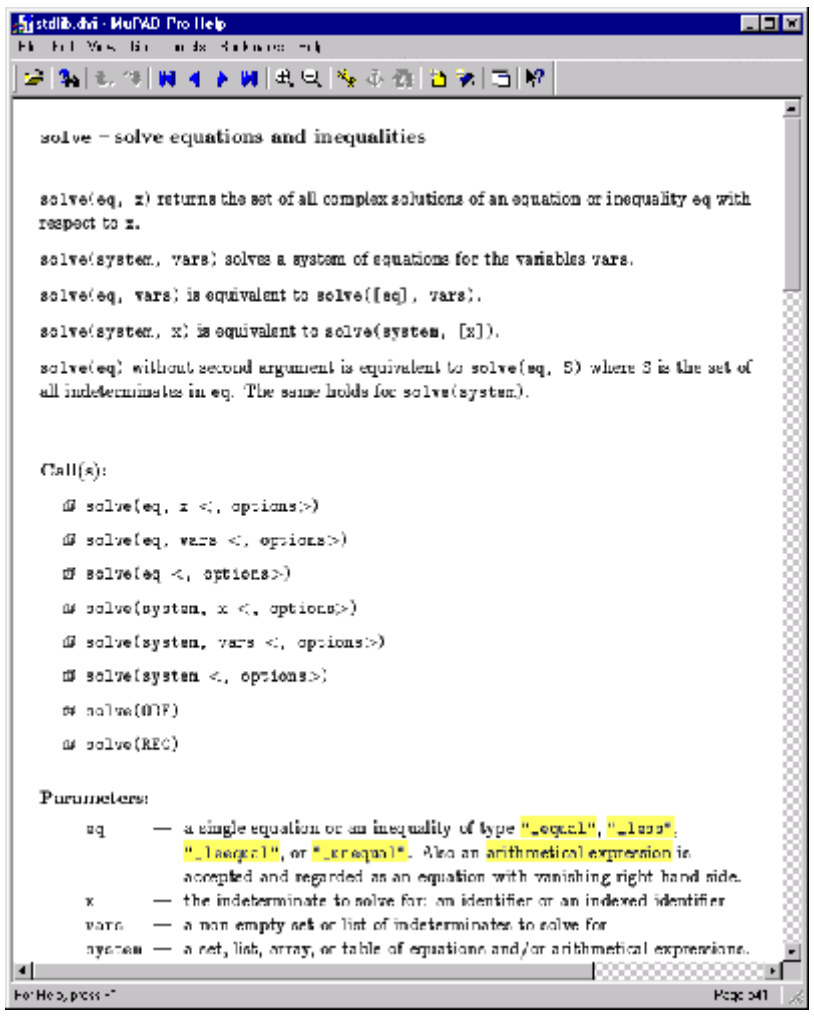
A função da página de ajuda correspondente provê informação mais detalhada, você pode consultar por entrar help (“nome”), onde nome é o nome da função. A função help opera seu argumento para ser uma corda, a qual é ou gerada por duplas de citações “ em MuPAD (Seção 4.11). O operador ? é uma forma de abreviação de help. Ele é usado sem parêntesis ou aspas:

```
>> ?solve
```

O layout das páginas de ajuda depende da versão do MuPAD. No seguinte exemplo, você pode ver uma página de ajuda no formato ASCII, como ele é retornado pela versão terminal do MuPAD em resposta a ?solve:

```
solve -- solve equations and inequalities

Introduction

 solve(eq, x) returns the set of all complex solutions of an equation or
 inequality eq with respect to x.

 solve(system, vars) solves a system of equations for the variables vars.

 solve(eq, vars) is equivalent to solve([eq], vars).

 solve(system, x) is equivalent to solve(system, [x]).

 solve(eq) without second argument is equivalent to solve(eq, S) where S
 is the set of all indeterminates in eq. The same holds for
solve(system).

Call(s)

 solve(eq, x <, options>)
 solve(system, vars <, options>)
 solve(eq, vars <, options>)
 solve(system, x <, options>)
 solve(eq <, options>)
 solve(system <, options>)
 solve(ODE)
 solve(REC)

Parameters

 eq      - a single equation or an inequality of type "_equal",
           "_less", "_leequal", or "_unequal". Also an arithmetical
           expression is accepted and regarded as an equation with
           vanishing right hand side.
 x       - the indeterminate to solve for:  an identifier or an indexed
           identifier
 vars    - a non-empty set or list of indeterminates to solve for
 system  - a set, list, array, or table of equations and/or
           arithmetical expressions. Expressions are regarded as
           equations with vanishing right hand side.
 ODE     - an ordinary differential equation:  an object of type ode.
 REC     - a recurrence equation:  an object of type rec.
...
```

Nós omitimos o restante da saída para salvar espaço. A seguinte figura mostra uma parte do hipertexto correspondente documento que aparece se você tem uma interface gráfica de usuário.



O sistema de ajuda é um sistema de hipertexto. Palavras-chave ativas são sublinhadas ou moldadas. Se você clicar nelas, você obtêm informações futuras sobre a noção correspondente. Os exemplos nas páginas de ajuda podem ser transferidos a janela de entrada do MuPAD clicando nos prompts sublinhados ou moldados correspondentes. Usuários de Windows usam o duplo clique ou arrastar e soltar.

Exercício 2.1: Descubra como usar o MuPAD diferenciador diff, e compute a quinta derivada de sin (x^2).
Solução A página de ajuda ?diff diz a você como computar derivadas de alta ordem:

```
>> diff(sin(x^2), x, x, x, x, x)
```

$$32\,x^5\cos\left(x^2\right) - 120\,x\cos\left(x^2\right) + 160\,x^3\sin\left(x^2\right)$$

Você pode também usar o extenso comando diff(diff(...,x),x).

2.2 Computando com números

Para computar com números, você pode usar o MuPAD como uma calculadora de bolso. O resultado da seguinte entrada é um número racional

```
>> 1 + 5/2
```

$$\frac{7}{2}$$

Você vê que MuPAD retorna resultados exatos (ou então arredondamentos de pontos flutuantes) quando computa com integrais e números racionais.

```
>> (1 + (5/2*3))/(1/7 + 7/9)^2
```

$$\frac{67473}{6728}$$

O símbolo ^ representa potenciação. MuPAD pode computar grandes números eficientemente. O peso do número q eu você pode computar é apenas limitado pelo espaço principal reconhecível. Por exemplo, a 123 potência de 1234 é uma integral distanciadamente grande.

```
>> 1234^123
170515806212727042875059727620626282654302313111068 29\
047052961932218391383486800747136630671706059857264 15\
923145543459005705896706714997090861025399048465147 93\
135617305563669993950104622035682027355757755070083 23\
844414777839602638706704268570040400328704248063968 06\
968655878650166993838833888319804591599428453724146 01\
809429717726107628595243406801014418529766279838067 20\
3562799104
```

Além das funções básicas de aritmética, MuPAD provê uma variedade de funções operando em números. Um simples exemplo é o fatorial n!=1,2... n de uma integral não negativa, que pode ser inserida em notações matemáticas.

```
>> 100!
933262154439441526816992388562667004907159682643816 21\
468592963895217599993229915608941463976156518286253 69\
792082722375825118521091686400000000000000000000000 00
```

A função isprime checa se uma integral positiva é primo. E aí retorna com TRUE ou FALSE.

```
>> isprime(123456789)
```

FALSE

Usando ifactor, você pode obter a fatoração em números primos:

```
>> ifactor(123456789)
```

$$3^2 \cdot 3607 \cdot 3803$$

2.2.1 Computações exatas

Agora supondo que queiramos computar o número √56. O problema é que o valor desse número é irracional, não pode ser escrito como um coeficiente numerador denominador de duas integrais exatas. Assim “computação” pode apenas significar encontrar uma representação exata que tão simples quanto possível. Quando você entra √56 via sqrtr, MuPAD retorna o seguinte:

```
>> sqrt(56)
```

$$2\sqrt{14}$$

O resultado da simplificação de √56 é o valor exato 2.√14. No MuPAD, √14(ou as vezes, 14^(1/2)) representa a solução positiva da equação x^2=14. Realmente, esta é a representação mais simples do resultado. Nós acentuamos que √14 é um objeto genuíno do MuPAD com certas propriedades (p.e, que este quadrado pode ser simplificado para 14). O sistema aplica-os automaticamente quando computa com tais objetos. Por exemplo:

```
>> sqrt(14)^4
```

196

Como outro exemplo de computação exata, deixe-nos determinar o limite

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n.$$

Nós usaremos a função limit e o símbolo infinity:

```
>> limit((1 + 1/n)^n, n = infinity)
```

e

Para entrar com este número na entrada do MuPAD, você tem que usar E ou exp(1), onde exp representa a função exponencial. MuPAD reconhece as regras exatas da manipulação para esse objeto. Por exemplo,usando o logaritmo natural ln nós encontramos:

```
>> ln(1/exp(1))
    -1
```

Nós vamos encontrar mais computações exatas depois neste tutorial

2.2.2 Aproximações numéricas

Além de computações exatas, MuPAD também pode executar aproximações numéricas. Por exemplo, você pode usar a função do MuPAD float para encontrar uma aproximação decimal para √56. Esta função computa o valor deste argumento em representação de pontos flutuantes:

```
>> float(sqrt(56))
    7.483314774
```

A precisão da aproximação depende do valor da variável local DIGITS, que determina o número de dígitos decimais para computações numéricas. Este valor que falta é 10:

```
>> DIGITS; float(67473/6728)
    10

    10.02868609
```

Variáveis globais como DIGITS afetam o comportamento do MuPAD e são também chamadas de variáveis ambientes. Você encontra uma lista completa de todas as variáveis ambientes na Seção “Environment Variables” do guia rápido de referência do MuPAD [Oev 03]. A variável DIGITS pode assumir qualquer valor integral entre 1 e 2^32-1:

```
>> DIGITS := 100: float(67473/6728); DIGITS := 10:
   10.028686087990487514863258026159334126040428061831l5\
   338882282996432818073721759809750297265160523187
```

Nós reajustamos o valor de DIGITS para 10 para a seguinte computação. Isso também pode ser alcançado via o comando delete DIGITS. Para operações aritméticas com números, MuPAD automaticamente usa computações aproximadas assim que pelo menos um dos números envolvidos é um valor de ponto flutuante:

```
>> (1.0 + (5/2*3))/(1/7 + 7/9)^2
    10.02868609
```

Por favor, note que nenhum dos dois chama

```
>> 2/3*sin(2), 0.6666666666*sin(2)
```

resultados são uma computação aproximada de sin(2), desde que, tecnicamente, sin(2) é uma expressão representando o valor exato de sin(2) e não um número:

$$\frac{2\sin(2)}{3},\ 0.6666666666\sin(2)$$

(A separação de ambos os valores por uma vírgula gera um tipo de dado especial, chamado sequência, que é descrito na Seção 4.5). Você tem que usar a função float para computar aproximações de pontos flutuantes das expressões acima.

```
>> float(2/3*sin(2)), 0.6666666666*float(sin(2))
    0.6061982846, 0.6061982845
```

A maioria das funções aritméticas no MuPAD, como sqrt, a função trigonométrica, a função exponencial, ou a logarítmica, automaticamente retorna valores aproximados onde seus argumentos é um número ponto flutuante:

```
>> sqrt(56.0), sin(3.14)
    7.483314774, 0.001592652916
```

A constante π e e são denotadas por PI e E=exp(1), respectivamente. MuPAD pode executar computações matemáticas com eles.

```
>> cos(PI), ln(E)
    -1, 1
```

Se desejar, você pode obter aproximações numéricas dessas constantes aplicando float:

```
>> DIGITS := 100: float(PI); float(E); delete DIGITS:
   3.14159265358979323846264338327950288419716939937510l\
   82097494459230781640628620899862803482534211706 8

   2.71828182845904523536028747135266249775724709369995 9\
   57496696762772407663035354759457138217852516642 7
```

Exercício 2.2: Compute √27-2√3 e cos (π/8) exatamente Determine a aproximação numérica com a precisão de 5 dígitos

Respostas: As representações exatas são:

>>sqrt(27-2*sqrt(3), cos(PI/8)

$$\sqrt{3},\ \frac{\sqrt{\sqrt{2}+2}}{2}$$

As aproximações numéricas são:

```
>>DIGITS := 5:
>>float(sqrt(27) – 2*sqrt(3)), float(cós(PI/8))
   1.7321, 0.92388
```

Elas estão corretas dentro de 5 dígitos.

- **Números complexos**

A unidade imaginária $\sqrt{-1}$ é representada no MuPAD pelo símbolo I na entrada e uma vertical i na saída composta:

```
>> sqrt(-1), I^2
```
$$\mathrm{i},\ -1$$

Você pode inserir números complexos no MuPAD na notação matemática usual x + yi. Ambas as partyes real de x e imaginária podem ser integrais, números racionais, ou números de pontos flutuantes:

```
>> (1 + 2*I)*(4 + I), (1/2 + I)*(0.1 + I/2)^3
```
$$2+9\,\mathrm{i},\ 0.073-0.129\,\mathrm{i}$$

Se você usar expressões simbólicas como, por exemplo, sqrt(2), MuPAD pode não retornar o resultado der um cálculo nas coordenadas cartesianas.

```
>> 1/(sqrt(2) + I)
```
$$\frac{1}{\sqrt{2}+\mathrm{i}}$$

A função rectform (atalho para forma retangular) assegura que o resultado está dividido dentro de partes reais e imaginárias:

```
>> rectform(1/(sqrt(2) + I))
```
$$\frac{\sqrt{2}}{3}-\frac{\mathrm{i}}{3}$$

As funções Re e Im retornam a parte real de x, e a parte imaginária de y respectivamente, de um número complexo x+yi. As funções do MuPAD conjugate e abs computam a conjugação complexa x-yI e o valor absoluto |x+yi|= $\sqrt{x2 + y2}$, respectivamente:

```
>> Re(1/(sqrt(2) + I)), Im(1/(sqrt(2) + I)),
   abs(1/(sqrt(2) + I)), conjugate(1/(sqrt(2) + I)),
   rectform(conjugate(1/(sqrt(2) + I)))
```
$$\frac{\sqrt{2}}{3},\ -\frac{1}{3},\ \frac{\sqrt{3}}{3},\ \frac{1}{\sqrt{2}-\mathrm{i}},\ \frac{\sqrt{2}}{3}+\frac{\mathrm{i}}{3}$$

- **Computação simbólica**

Esta sessão compreende alguns exemplos das sessões do MuPAD que ilustram uma pequena seleção do poder do sistema de manipulação simbólica. O conhecimento matemático está contido essencialmente nas funções do MuPAD para diferenciação, integração, simplificação de expressões, etc. Esta demonstração não procede em uma maneira particularmente sistemática: nós aplicamos as funções do sistema para objetos de vários tipos, tipos sequências, jogos, listas, expressões, etc. Isto está explicado em detalhes passo a passo no Capítulo 4.

- **Exemplos introdutórios**

Uma expressão simbólica no MuPAD pode conter quantidades indeterminadas (ou identificadores). A seguinte expressão contém duas incógnitas x e y.

```
>> f := y^2 + 4*x + 6*x^2 + 4*x^3 + x^4
```
$$x^4+4\,x^3+6\,x^2+4\,x+y^2$$

Usando o operador tarefa:=, nós temos tarefa e expressão para um identificador f, que agora pode ser usado como uma abreviação para a expressão. Nós dizemos que o que vem depois é o valor da identidade f. Nós notamos que MuPAD troca a ordem dos termos.

MuPAD oferece a função do sistema diff para expressões diferenciais:

```
>> diff(f, x), diff(f, y)
```
$$4\,x^3+12\,x^2+12\,x+4,\ 2\,y$$

Aqui, nós computamos ambas as derivadas com respeito a x e a y. Você pode obter altas derivações cada uma aninhada as chamadas de diff, ou por uma simples chamada:

```
>> diff(diff(diff(f, x), x), x), diff(f, x, x, x)
```
$$24\,x+24,\ 24\,x+24$$

Alternativamente, você pode usar o operador diferencial ‘, que mapeia uma função e sua derivada:

```
>> sin', sin'(x)
```
$$\cos,\ \cos(x)$$

O símbolo ‘ para derivada é uma forma de atalho do operador diferencial D. A chamada D (função) retorna a derivada

```
>> D(sin), D(sin)(x)
```
$$\cos,\ \cos(x)$$

Você pode computar integrais usando o comando int. O seguinte comando é uma integral definitiva no intervalo real entre 0 e 1:

```
>> int(f, x = 0..1)
```
$$y^2 + \frac{26}{5}$$

O próximo comando determina uma integral indefinida e retorna uma expressão contendo a integração variável x e o parâmetro simbólico y.

```
>> int(f, x)
```
$$\frac{x^5}{5} + x^4 + 2\,x^3 + 2\,x^2 + x\,y^2$$

Note que int retorna uma antiderivada especial, não uma geral (com constante aditiva).
Se você tentar computar uma integral indefinida de uma expressão e esta não puder ser representada por funções elementares, então int retorna a chamada simbolicamente:

```
>> integral := int(1/(exp(x^2) + 1), x)
```
$$\int \frac{1}{e^{x^2}+1}\,dx$$

Não obstante, este objeto tem propriedades matemáticas. O diferenciador reconhece que sua derivada é a integral:

```
>> diff(integral, x)
```
$$\frac{1}{e^{x^2}+1}$$

Integrais definidas também podem ser retornadas simbolicamente por int:

```
>> int(1/(exp(x^2) + 1), x = 0..1)
```
$$\int_0^1 \frac{1}{e^{x^2}+1}\,dx$$

O objeto matemático correspondente é um número real e a saída é uma representação exata deste número do qual o MuPAD foi incapaz de simplificar mais adiante. Como usualmente, você pode obter uma aproximação de pontos flutuantes aplicando float:

```
>> float(%)
```
$$0.41946648$$

O símbolo % (que é equivalente a last (1)) é uma abreviação para a expressão previamente computada (Capítulo 12).
MuPAD conhece as funções matemáticas mais importantes como a raiz quadrada sqrt, a função exponencial exp, as funções trigonométricas sin, cos e tan, as funções hiperbólicas sinh, c osh, tanh, as funções inversamente correspondentes ln, arc, arcos, arctan, arcsinh, arccosh, arctanh, tão bem como uma variedade de outras funções especiais como, por exemplo, a função gamma, a função erro erf, Funções de Bessel, etc. (Seção Funções “Funções matemáticas especiais do Guia rápido de referência do MuPAD dá um guia de sobrevivência.) Particularmente, MuPAD reconhece as regras de manipulação para essas funções (por exemplo, a adição de teoremas para as funções trigonométricas) e aplica isso. Pode computar aproximações de pontos flutuantes como, por exemplo., float (exp(1))= 2,718.., e conhece os valores especiais (por exemplo) sin (PI)= 0). Se você chama essas funções, muitas vez elas retornam por si mesmas simbólicamente, desde que elas sejam a representação exata mais simples do valor correspondente.

```
>> sqrt(2), exp(1), sin(x + y)
```
$$\sqrt{2},\ e,\ \sin(x+y)$$

Para muitos usuários, a principal característica do sistema é simplificar ou transformar como expressões usando as regras da computação. Por exemplo, a função expand do sistema “expande” funções como exp, sin, etc. Por simplificar da adfição de teoremas de seus argumentos como uma soma simbólica:

```
>> expand(exp(x + y)), expand(sin(x + y)),
   expand(tan(x + 3*PI/2))
```
$$e^x\,e^y,\ \cos(x)\sin(y) + \cos(y)\sin(x),\ -\frac{1}{\tan(x)}$$

Geralmente falando, uma das tarefas principais de um sistema algébrico computacional é manipular e simplificar expressões. Por trás de expand, MuPAD provê a função collect, combine, factor, normal, partrfrac, radsimp, rewrite, e simplify, para manipulação. Estas são apresentadas mais detalhadamente no Capítulo 9. Nós mencionamos brevemente alguns deles no que se segue.
A função normal encontra um denominador comum para expressões racionais:

```
>> f := x/(1 + x) - 2/(1 - x): g := normal(f)
```

$$\frac{x^2 + x + 2}{x^2 - 1}$$

Ainda mais, normal automaticamente cancela fatores comuns no numerador e no denominador:

```
>> normal(x^2/(x + y) - y^2/(x + y))
```

$$x - y$$

Reciprocamente, partfrac (abreviado/ atalho para “fração parcial”) decompõe uma expressão racional dentro de uma soma de termos racionais com denominadores simples:

```
>> partfrac(g, x)
```

$$\frac{2}{x-1} - \frac{1}{x+1} + 1$$

A função simplify é a simplificação universal e tenta encontrar uma representação que é tão simples quanto possível:

```
>> simplify((exp(x) - 1)/(exp(x/2) + 1))
```

$$e^{\frac{x}{2}} - 1$$

Você pode manipular a simplificação usando simplify com argumentos adicionais (veja?simplify). A função radsimp simplifica expressões aritméticas contendo radicais (raízes):

```
>> f := sqrt(4 + 2*sqrt(3)): f = radsimp(f)
```

$$\sqrt{\sqrt{3}+2}\sqrt{2} = \sqrt{3}+1$$

Aqui nós geramos uma equação, da qual é um assunto genuíno do MuPAD.
Outra função importante é factor, pela qual decompõe uma expressão dentro de um produto de uns simples.

```
>> factor(x^3 + 3*x^2 + 3*x + 1),
   factor(2*x*y - 2*x - 2*y + x^2 + y^2),
   factor(x^2/(x + y) - z^2/(x + y))
```

$$(x+1)^3,\ (x+y-2)\cdot(x+y),\ \frac{(x-z)\cdot(x+z)}{(x+y)}$$

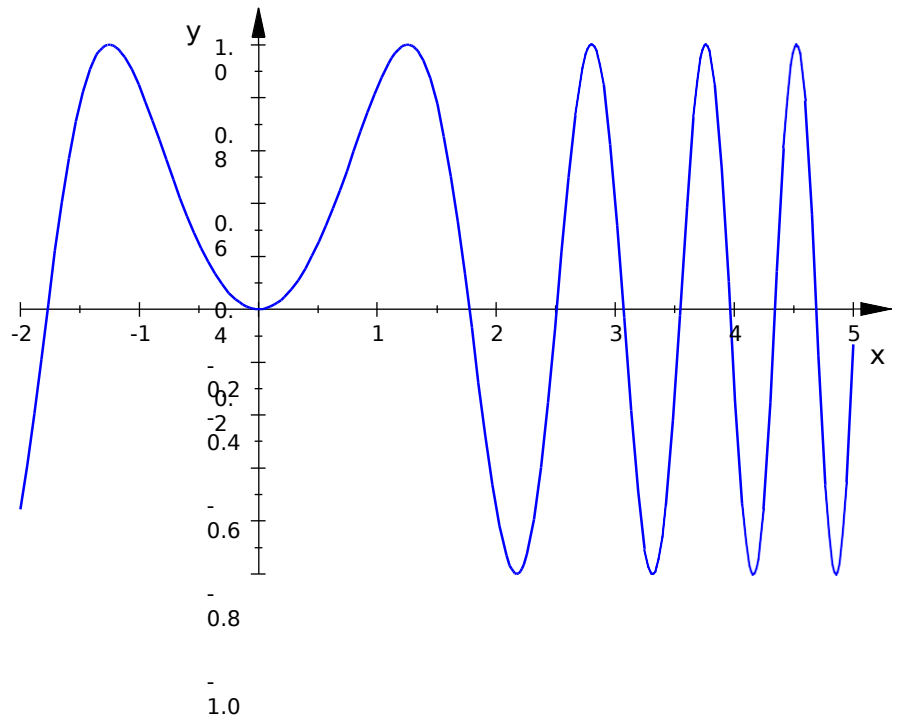
A função limit faz o que o nome sugere. Por exemplo, a função sin(x)/x têm um polo removível em x=D. Este limite para x->0 é 1:

```
>> limit(sin(x)/x, x = 0)
```

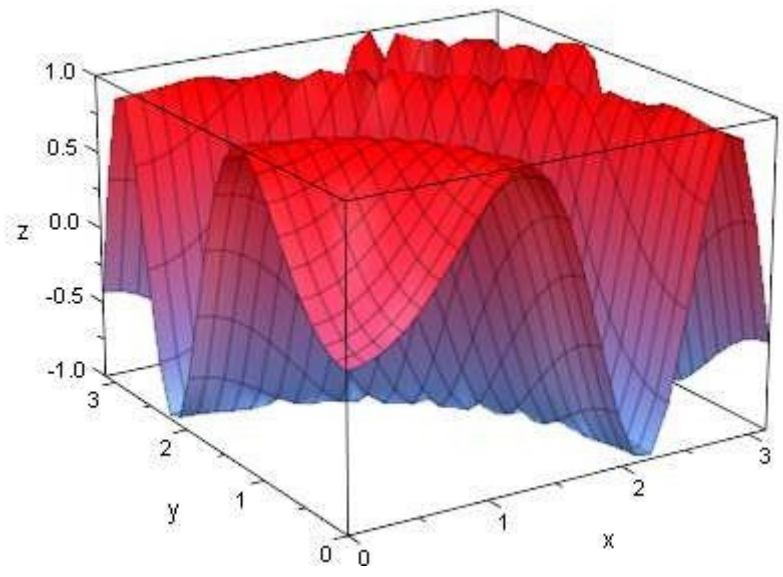
$$1$$

Numa sessão do MuPAD, você pode definir funções próprias em seus muitos caminhos. Um método simples e intuitivo é usar o operador seta -> (o símbolo de menos seguido deste símbolo “maior que”):
No Capítulo 18, nós discutimos as características programadas e descrevemos como implementar algoritmos mais complexos como procedimentos do MuPAD.

Nas janelas base das versões do MuPAD, você pode usar as facilidades gráficas para visualizar objetos matemáticos imediatamente. As funções relevantes do MuPAD para gerar gráficos são plotfunc2d,, plotfunc3d, e as rotinas da biblioteca de gráficos plot. Você pode deixar o MuPAD desenhar os gráficos ou funções com um ou dois argumentos usando plotfunc2d ou plotfunc3d, respectivamente.
>>plotfunc2d(sin(x^2),x=-2..5)



>>plotfunc3d(sin(x^2+y^2),x=0..PI,y=0..PI)



Dependendo de sua versão do MuPAD ou o módulo de gráficos abre uma janela separada, ou o enredo aparece no caderno em baixo da chamada de comando de gráficos. Você pode manipular os gráficos interativamente. Você encontra uma descrição das características gráficas no Capítulo 11.

Resolver equações ou sistemas de equações é certamente a tarefa mais importante para um sistema algébrico computacional. Isso pode ser feito via comando solve no MuPAD:

```
>> equations := {x + y = a, x - a*y = b}:
>> unknowns := {x, y}:
>> solve(equations, unknowns, IgnoreSpecialCases)
```

$$\left\{\left[x = \frac{a^2+b}{a+1}, y = \frac{a-b}{a+1}\right]\right\}$$

Aqui, nós geramos um jogo de duas equações e um jogo de incógnitas que gostaríamos de resolver. MuPAD retorna o resultado em termos de equações simplificadas das quais você pode ler a saída da solução. No exemplo acima, há dois parâmetros mais simbólicos a e b. Isto é porque nós temos mantido o comando solve do qual nós tivemos mantido os símbolos a ser expressos em tremos dos outros, a "opção" IgnoreSpecialCases avisa o MuPAD para ignorar a possibilidade que a possa ser -1, onde a solução acima esteja incorreta. Sem esta opção, MuPAD retorna uma solução completa com três fileiras:

```
>> solve(equations, unknowns)
```

$$\begin{cases} \left\{\left[x = \frac{a^2+b}{a+1}, y = \frac{a-b}{a+1}\right]\right\} & \text{if} \quad a \neq -1 \\ \{[x = -y - 1]\} & \text{if} \quad a = -1 \wedge b = -1 \\ \emptyset & \text{if} \quad b \neq -1 \wedge a = -1 \end{cases}$$

No seguinte exemplo, nós temos apenas uma equação em uma incógnita. MuPAD automaticamente reconhece a incógnita e a resolve:

```
>> solve(x^2 - 2*x + 2 = 0)
```

$$\{[x = 1 - \mathrm{i}], [x = 1 + \mathrm{i}]\}$$

Se nós fornecermos a incógnita x para resolvê-la, o formato da saída muda.

```
>> solve(x^2 - 2*x + 2 = 0, x)
```

$$\{1 - \mathrm{i}, 1 + \mathrm{i}\}$$

O resultado é um jogo contendo duas (complexas) soluções de equações quadráticas. Você encontra maiores detalhes do comando solve no Capítulo 8.

A função sum e product manipula somas simbólicas e produtos. Por exemplo, a soma bem conhecida 1 + 2 +...+ n é:

```
>> sum(i, i = 1..n)
```

$$\frac{n\,(n+1)}{2}$$

O produto 1 . 2...n é conhecido como um fatorial n!:

```
>> product(i^3, i = 1..n)
```

$$n!^3$$

Existe muitas estruturas de dados para vetores e matrizes no MuPAD. No princípio, você pode usar variedades (Seção 4.9) para representar tais objetos. Entretanto, isso é tão mais intuitivo para trabalhar com o tipo de dado "matrix". Você pode gerar matrizes usando a função do sistema matrix:

```
>> A := matrix([[1, 2], [a, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ a & 4 \end{pmatrix}$$

Assuntos de matrizes construídas nesse capítulo tem a propriedade conveniente que as operações aritméticas básicas +, *, etc . é especializado ("sobrecarregado") de acordo com o contexto matemático apropriado. Por exemplo, você pode usar + ou * para adicionar ou multiplicar matrizes, respectivamente (se as dimensões partem):

```
>> B := matrix([[y, 3], [z, 5]]):
>> A, B, A + B, A*B
```

$$\begin{pmatrix} 1 & 2 \\ a & 4 \end{pmatrix}, \begin{pmatrix} y & 3 \\ z & 5 \end{pmatrix}, \begin{pmatrix} y+1 & 5 \\ a+z & 9 \end{pmatrix}, \begin{pmatrix} y+2\,z & 13 \\ 4\,z+a\,y & 3\,a+20 \end{pmatrix}$$

A potência A^(-1), equivalente a 1/A, denota a matriz invertida:

```
>> A^(-1)
```

$$\begin{pmatrix} -\frac{2}{a-2} & \frac{1}{a-2} \\ \frac{a}{2\,a-4} & -\frac{1}{2\,a-4} \end{pmatrix}$$

A função linalg::det, da biblioteca linalg do MuPAD para álgebra linear (Secão 4.15.4), computa o determinante:

```
>> linalg::det(A)
```

$$4-2\,a$$

Vetores da coluna de dimensões n podem ser interpretados como matrizes n X 1:

```
>> b := matrix([1, x])
```

$$\begin{pmatrix} 1 \\ x \end{pmatrix}$$

Você pode confortavelmente determinar a solução $A^{-1}\vec{b}$ do sistema de equações lineares $A\vec{x} = \vec{b}$, com o coeficiente acima da matriz A e o previamente definido o coeficiente acima da matriz A e o previamente definido $\underset{b}{\rightarrow}$ no lado à direita:

```
>> solutionVector := A^(-1)*b
```

$$\begin{pmatrix} \frac{x}{a-2} - \frac{2}{a-2} \\ \frac{a}{2\,a-4} - \frac{x}{2\,a-4} \end{pmatrix}$$

Agora você pode aplicar a função normal para cada componente do vetor pelos sentidos da função map do sistema, assim simplificando a representação

```
>> map(%, normal)
```

$$\begin{pmatrix} \frac{x-2}{a-2} \\ \frac{a-x}{2\,a-4} \end{pmatrix}$$

Para verificar a computação do MuPAD, você pode multiplicar o vetor solução pela matriz A:

```
>> A * %
```

$$\begin{pmatrix} \frac{2\,(a-x)}{2\,a-4} + \frac{x-2}{a-2} \\ \frac{4\,(a-x)}{2\,a-4} + \frac{a\,(x-2)}{a-2} \end{pmatrix}$$

Depois da simplificação, você pode checar que o resultado é igual b:

```
>> map(%, normal)
```

$$\begin{pmatrix} 1 \\ x \end{pmatrix}$$

Seção 4.15 provê mais informação em controle de matrizes e vetores.

Exercício 2.3: Compute uma forma expandida da expressão $(x^2+y)^5$.

$$\frac{x^2-1}{x+1} = x-1$$

Exercício 2.4: Use MuPAD para checar que controla .

Exercício 2.5: gere um plot da fun cão 1/sin(x) para 1≤ x≤ 10.

Exercício 2.6: Obtenha informações detalhadas sobre a função limit para verificar os seguintes limites

$$\lim_{x\to 0} \frac{\sin(x)}{x} = 1\,, \quad \lim_{x\to 0} \frac{1-\cos(x)}{x} = 0\,, \quad \lim_{x\to 0+} \ln(x) = -\infty\,,$$

$$\lim_{x\to 0} x^{\sin(x)} = 1\,, \quad \lim_{x\to \infty} \left(1+\frac{1}{x}\right)^x = e\,, \quad \lim_{x\to \infty} \frac{\ln(x)}{e^x} = 0\,,$$

$$\lim_{x\to 0+} x^{\ln(x)} = \infty\,, \quad \lim_{x\to \infty} \left(1+\frac{\pi}{x}\right)^x = e^{\pi}\,, \quad \lim_{x\to 0-} \frac{2}{1+e^{-1/x}} = 0\,.$$

O limite lim $e^{\cos(x)}$ não existe. Como o MuPAD reage? x→0

Exercício 2.7: Obtenha uma informação detalhada sobre a função sum. A chamada sum (f (k), k=a..b) computa uma fórmula fechada de uma soma finita ou infinita, se possível. Use o MuPAD para verificar a seguinte identidade:

$$\sum_{k=1}^{n}(k^2+k+1)=\frac{n\,(n^2+3\,n+5)}{3}\,.$$

Determine os valores das seguintes séries:

$$\sum_{k=0}^{\infty}\frac{2\,k-3}{(k+1)\,(k+2)\,(k+3)}\,,\quad \sum_{k=2}^{\infty}\frac{k}{(k-1)^2\,(k+1)^2}\,.$$

Exercício 2.8: Compute 2 . (A + B), A . B, e $(A - B)^{-1}$ para as seguintes matrizes:

$$A=\begin{pmatrix}1&2&3\\4&5&6\\7&8&0\end{pmatrix},\quad B=\begin{pmatrix}1&1&0\\0&0&1\\0&1&0\end{pmatrix}.$$

2.3.2 **Esboço de curva**

Na próxima sessão de demonstração, nós usamos alguns dos sistemas de funções da sessão anterior para esboçar e discutir a curva dada pela função racional

$$f:\ x\ \mapsto \frac{(x-1)^2}{x-2}+a$$

Com um parâmetro a. Primeiro, nós determinamos algumas características desta função: descontinuidades, valores extremos, e comportamento para o largo x.

```
>> f := x -> ((x - 1)^2/(x - 2) + a):
>> singularities := discont(f(x), x)
```

$\{2\}$

A função discont determina a descontinuidade da função f com relação a variável x. Ela retorna um jogo de pontos parecidos. Assim, o f abaixo é definido e contínuo para todo x ≠ 2. Obviamente, x = 2 é um polo. Realmente, MuPAD encontra o limite $\mp\infty$ quando você aproxima este ponto da esquerda para a direita, respectivamente:

```
>> limit(f(x), x = 2, Left), limit(f(x), x = 2, Right)
```

$-\infty,\ \infty$

Você pode encontrar as raízes de *f* por resolver a equação *f* = 0:

```
>> roots := solve(f(x) = 0, x)
```

$$\left\{1-\frac{\sqrt{a\,(a+4)}}{2}-\frac{a}{2},\frac{\sqrt{a\,(a+4)}}{2}-\frac{a}{2}+1\right\}$$

Dependendo em *a*, cada de ambos ou nenhuma das raízes são reais. Agora, nós queremos encontrar o local extrema de *f*. Para este fim, nós determinamos as raízes da primeira derivada *f*':

```
>> f'(x)
```

$$\frac{2\,x-2}{x-2}-\frac{(x-1)^2}{(x-2)^2}$$

```
>> extrema := solve(f'(x) = 0, x)
```

$\{1,3\}$

Estes são os candidatos para local extrema. Contudo, alguns destes meios podem ser pontos selados. Se a segunda derivada *f*'' não desaparecer nesses pontos, então ambos são realmente extrema. Nós conferimos:

```
>> f''(1), f''(3)
```

$-2,\ 2$

Nossos resultados implicam que *f* tem as seguintes propriedades: para qualquer escolha do parâmetro *a*, há um local máximo em *x* = 1, um polo em *x* = 2, e um local mínimo em *x* = 3. Os valores correspondentes de *f* nestes pontos são

```
>> maxvalue := f(1); minvalue := f(3)
```

a

$a+4$

f tende para $\mp\infty$ para $x \to \mp\infty$:

```
>> limit(f(x), x = -infinity), limit(f(x), x = infinity)
```

$-\infty,\ \infty$

Nós podemos especificar o comportamento de *f* mais precisamente para grandes valores de *x*. Isso assintomaticamente aproxima a função linear
$x \mapsto x + a$:

```
>> series(f(x), x = infinity)
```

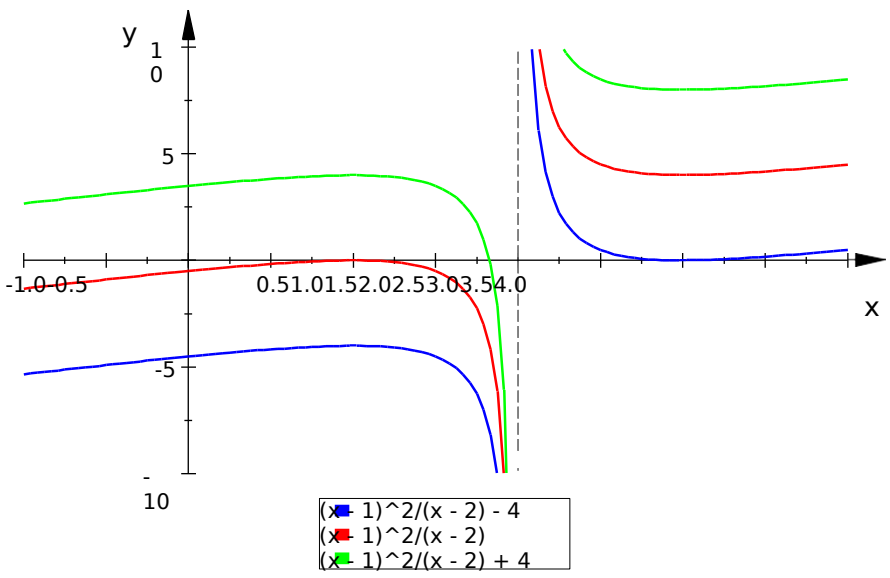
$$x + a + \frac{1}{x} + \frac{2}{x^2} + \frac{4}{x^3} + \frac{8}{x^4} + O\left(\frac{1}{x^5}\right)$$

Aqui nós empregamos a função series para computar uma expansão assintomática de *f* (Seção 4.13). Nós podemos facilmente checar nossos resultados visualmente plotando o gráfico de *f* para vários valores de *a*:

```
>> F := subs(f(x), a = -4): G := subs(f(x), a = 0):
   H := subs(f(x), a = 4): F, G, H
```

$$\frac{(x-1)^2}{x-2} - 4,\ \frac{(x-1)^2}{x-2},\ \frac{(x-1)^2}{x-2} + 4$$

A função subs (Capítulo 6) substitui subexpressões: no exemplo, nós substituímos os valores concretos -4, 0 e 4, respectivamente, para *a*. Nós agora podemos plotar as três funções juntas em uma imagem:
>> plotfunc2d(F, G, H, x = -1..4)



2.3.3 Teoria elementar dos números

MuPAD provê um lote de teoria elementar dos números em funções teóricas, por exemplo:

. isprime (n) testa se n ∈ ℕ é um número primo,

. ithprime (n) retorna o enésimo número primo,

.nextprime (n) encontra o menor número primo ≥ n,

ifactor (n) computa a fatoração prima de n.

Essas rotinas são muito rápidas. Contudo, desde que elas empreguem testes probabilísticos de números primos, eles podem retornar resultados errados com uma probabilidade muito pequena. Ao invés de isprime, você pode usar a função (mais lenta) numlib::proveprime com um erro-livre de teste de primariedades.

Deixe-nos gerar uma lista de todos os primos até 10 000. Aqui é um dos muitos modos de fazer isto:

```
>> primes := select([$ 1..10000], isprime)
       [2, 3, 5, 7, 11, 13, 17, ..., 9949, 9967, 9973]
```

Primeiro, nós geramos a sequência de todas as integrais positivas até 10 000 pelos sentidos do gerador de sequência $ (Seção 4.5). O quadrado de colchetes [] converte isto para um a lista do MuPAD. Então select (Seção 4.6) elimina todos estes elementos de listas para as quais a função isprime, é provida como um segundo argumento, retorna FALSE. O número destes primos é igual ao número de elementos de lista, dos quais nós podemos obter via nops (“número de operadores,” Seção 4.1):

```
>> nops(primes)
    1229
```

Alternativamente, nós podemos gerar a mesma lista de números primos por:

```
>> primes := [ithprime(i) $ i = 1..1229]:
```

Aqui nós usamos o fato que nós já sabemos o número de primos até 10 000. Outra possibilidade é gerar uma larga lista de números primos e descartar as unidades que são maiores do que 10 000:

```
>> primes := select([ithprime(i) $ i=1..5000],
                    x -> (x<=10000)):
```

Aqui, o objeto x -> (x <=10000) representa a função que mapeia cada x para a inequação x <=10000). O comando select então guarda somente essa lista de elementos para cada inequações avaliadas para TRUE.

No próximo exemplo, nós usamos o loop rapeat (Capítulo 16) para gerar uma lista de primos. Com a ajuda do operador de concatenação . (Seção 4.6), nós sucessivamente juntamos primos *i* para uma lista até nextprime (i +1), o próximo primo maior que *i* excede 10 000. Nós começamos com a lista vazia e o primeiro primo *i* = 2:

```
>> primes := []: i := 2:
>> repeat
     primes := primes . [i];
     i := nextprime(i + 1)
   until i > 10000 end_repeat:
```

Agora, nós consideramos a famosa conjectura de Goldbach:

“Toda e qualquer integral maior que 2 é a soma de dois números primos.”

Nós podemos para verificar esta conjectura para todos os números até 10000. Primeiro, nós geramos a lista de pares integrais [4, 6, . . ., 10000]:

```
>> list := [2*i $ i = 2..5000]:
>> nops(list)
    4999
```

Agora, nós selecionamos estes números da lista que não pode ser escrita na forma “primo + 2”. Isto é feito para testar para cada *i* na lista que *i* – 2 é um primo:

```
>> list := select(list, i -> (not isprime(i - 2))):
>> nops(list)
    4998
```

A única integral que foi eliminada é 4 (desde que todas os outros pares positivos integrais *i* – 2 é igual e maior que 2, e consequentemente não é primo).
Agora nós descartamos todos os números da forma “primo + 3”:

```
>> list := select(list, i -> (not isprime(i - 3))):
>> nops(list)
    3770
```

As 3770 integrais restantes não são nenhuma delas da forma “primo + 2” nem da forma “primo + 3”. Nós agora continuamos este procedimento pelos significados de um loop while (Capítulo 16). No loop, *j* sucessivamente corre através de todos os números primos > 3, e os números da forma “ primo + j” são eliminados. Um comando print (Seção 13.1.1) produz o número de integrais restantes em cada passo. O loop termina assim que a lista se esvazia:

```
>> j := 3:
>> while list <> [] do
       j := nextprime(j + 1):
       list := select(list, i -> (not isprime(i - j))):
       print(j, nops(list)):
   end_while:
                          5, 2747

                          7, 1926

                          11, 1400

                             ...

                           163, 1

                           167, 1

                           173, 0
```

Assim nós confirmamos que a conjectura de Goldbach se mantém verdadeira para toda e qualquer integral positiva até 10 000. Nós mostramos até mesmo que todos esses números podem ser escritos como uma soma de um primo menor ou igual a 173 e outro primo.

No próximo exemplo, nós geramos uma lista de distâncias entre dois primos sucessivos até 500:

```
>> primes := select([$ 1..500], isprime):
>> distances := [primes[i] - primes[i - 1]
                 $ i = 2..nops(primes)]
   [1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6, 2,

      6, 4, 2, 6, 4, 6, 8, 4, 2, 4, 2, 4, 14, 4, 6, 2,

      10, 2, 6, 6, 4, 6, 6, 2, 10, 2, 4, 2, 12, 12, 4,

      2, 4, 6, 2, 10, 6, 6, 6, 2, 6, 4, 2, 10, 14, 4, 2,

      4, 14, 6, 10, 2, 4, 6, 8, 6, 6, 4, 6, 8, 4, 8, 10,

      2, 10, 2, 6, 4, 6, 8, 4, 2, 4, 12, 8, 4, 8]
```

A chamada indexada primes [i] retorna o enésimo elemento na lista.

A função zip (Seção 4.6) provê um método alternativo. A chamada zip (a, b, f) combina duas listas *a* = [a1, a2,...] e b = [b1, b2,...] modo de componente pelos sentidos da função *f*: a lista resultante é

$$[f(a_1, b_1), f(a_2, b_2), \ldots]$$

e tem como tantos elementos como o atalho das duas listas. Em nosso exemplo, nós aplicamos estes para a lista de primos *a* = [a1,...,an], a lista de primos "atalhada" b= [a2,...,an], e a função (x, y) ↣ y – x. Nós geramos primeiro uma cópia trocada de lista de primo deletando o primeiro elemento, assim de simplificando mais rápido a lista:

```
>> b := primes: delete b[1]:
```

O seguinte comando retorna o mesmo resultado como acima:

```
>> distances := zip(primes, b, (x, y) -> (y - x)):
```

Nós presenciamos outra função útil na seção 2.2, a rotina ifactor para fatorar uma integral dentro de números primos. A chamada ifactor (n) retorna um objeto do mesmo tipo como factor: isto é um tipo de dado especial Factored. Objetos deste tipo são impressos na tela em uma forma que é facilmente legível:

```
>> ifactor(-123456789)
     -3^2 · 3607 · 3803
```

Internamente, os fatores primos e os expoentes são armazenados na forma de uma lista, e você pode extrair eles pelo uso de op ou por um acesso indexado.
Consulte as páginas de ajuda de ifactor e Factored para detalhes. A lista interna tem o formato

$$[s, p_1, e_1, \ldots, p_k, e_k]$$

Com primos $p_1, \ldots, p_k$, seus expoentes $e_1, \ldots, e_k$, e o sinal $s = \pm 1$, tal qual $n = s.\ p_1^{e1}.p_2^{e2}...p_k^{ek}$:

```
>> op(%)
     -1, 3, 2, 3607, 1, 3803, 1
```

Agora nós empregamos a função ifactor para encontrar quantas integrais entre 2 e 10 000 são divisíveis por dois números primos exatamente distintos. Nós notamos que o objeto retornado por ifactor (n) tem 2m + 1 elementos em sua lista de representação, onde *m* é o número de números primos distintos divisores de *n*. Assim, a função

```
>> m := (nops@ifactor - 1)/2:
```

retorna o número de fatores primos distintos. O símbolo @ gera a composição (Seção 4.12) das duas funções ifactor e nops. Assim a chamada **m (k) = (nops (ifactor (k)) – 1) /2** para **k** se torna uma integral. Nós construímos a lista de valores (*m* (*k*) para *k* = 2, . . ., 10000:

```
>> list := [m(k) $ k = 2..10000]:
```

O seguinte loop for (Seção 16) apresenta o número de integrais com precisão *i* = 1, 2, . . ., 6 divisores primos distintos:

```
>> for i from 1 to 6 do
       print(i, nops(select(list, x -> (x = i))))
   end_for:
                          1, 1280

                          2, 4097

                          3, 3695

                           4, 894

                           5, 33

                            6, 0
```

Assim há 1280 integrais com exatamente um divisor primo no intervalo escaneado, 4097 integrais com precisamente dois fatores primos distintos, e assim vai. Isso é fácil de conferir como o intervalo não contém integrais com seis ou mais divisores primos: o menor tal número
2 . 3 . 5 . 7 . 11 . 13 = 30 030 excede 10 000.
A biblioteca numlib cumpre várias funções numéricas teóricas. Entre outros, ela contém a rotina numlib : : numberprimedivisors, equivalente para o **m** acima, computando o número de divisores primos. Nós nos referimos ao Capítulo 3 para uma descrição das bibliotecas do MuPAD;

Exercício 2.9: primos da forma $2^n \pm 1$ sempre produziram interesse particular.

a) primos da forma $2^p - 1$, onde p é um número primo, são chamados de primos de Mersenne. Encontre todos os primos de Mersenne para $1 < p \leq 1000$.

b) Para uma integral positiva n, o enésimo número de Fermat é $2(^{2n}) + 1$. Refute a conjectura de Fermat que todos esses números são números primos.

Capítulo 3

As bibliotecas do MuPAD

A maioria dos conhecimentos matemáticos do MuPAD é organizado em bibliotecas. Tais como uma biblioteca cumpre uma coleção de funções para resolver problemas em uma área particular, tais como álgebra linear, teoria dos números, analise numérica, etc. Funções de biblioteca são escritas na linguagem de programação do MuPAD. Você pode usar elas do mesmo modo como funções de kernel, sem conhecimento de nada sobre a linguagem de programação.

Seção “Bibliotecas e Módulos” do guia rápido de referência do MuPAD contém um guia de sobrevivência de todas as bibliotecas. Alternativamente, você pode conseguir uma lista de todas as bibliotecas disponíveis pela chamada info():

```
>> info()
   -- Libraries:
   Ax,         Cat,       Dom,       Graph,     RGB,
   Series,     Type,      adt,       combinat,  detools,
   fp,         generate,  groebner,  import,    intlib,
   linalg,     linopt,    listlib,   matchlib,  module,
   numeric,    numlib,    ode,       orthpoly,  output,
   plot,       polylib,   prog,      property,  solvelib,
   specfunc,   stats,     stdlib,    stringlib, student,
   transform
```

As bibliotecas são desenvolvidas continuamente, e em versões futuras do MuPAD vai prover funcionalidades adicionais.

Neste capítulo, nós não endereçamos a funcionalidade matemática de bibliotecas mas discutimos bastante seu uso geral.

3.1 Informações sobre uma biblioteca particular

Você pode obter informação e ajuda para bibliotecas pela chamada de funções info e help. A função info dá uma lista de todas as funções instaladas na biblioteca. A biblioteca numlib é uma coleção de funções teóricas de números:

```
>> info(numlib)
   Library 'numlib':     the package for elementary
   number theory

   -- Interface:
   numlib::Lambda,         numlib::Omega,
   numlib::contfrac,       numlib::cornacchia,
   numlib::decimal,        numlib::divisors,
   numlib::ecm,            numlib::factorGaussInt,
   numlib::fibonacci,      numlib::fromAscii,
   ...
```

Os comandos help ou ? supre uma descrição mais detalhada da biblioteca:

```
>> ?numlib
   numlib - library for number theory

   Table of contents

    o contfrac - the domain of continued fractions

    o decimal - infinite representation of rational numbers

    o divisors - divisors of an integer

    o ecm - factor an integer using the elliptic curve method

    o fibonacci - Fibonacci numbers

    o fromAscii - decoding of ASCII codes

    ...
```

Se você tiver uma interface gráfica de usuário, este comando abre uma janela de ajuda e você pode navegar para a página de ajuda de uma das funções listadas por clicar em seu nome com o mouse.

A função numlib::decimal da biblioteca numlib compujta a expansão decimal de um número racional:

```
>> numlib::decimal(123/7)
    17, [5, 7, 1, 4, 2, 8]
```

Como para outro sistema de funções, você pode requerer informação na biblioteca das funções pelos sentidos de help ou ?:

```
>> ?numlib::decimal
```

Você pode ter dado uma olhada na implementação de uma biblioteca de função pelo uso de expose:

```
>> expose(numlib::decimal)
   proc(a)
     name numlib::decimal;
     local p, q, s, l, i;
   begin
     if not testtype(a, Type::Numeric) then
        ...
   end_proc
```

3.2 Exportando bibliotecas

Você viu na seção anterior que a chamada de sintaxe para uma biblioteca de função é library::function, onde library e function são os nomes da biblioteca e a função, respectivamente. Por exemplo, a biblioteca numeric para computações numéricas contém a função numeric::fsolve. Ela implementa uma versão modificada do tão conhecido método de Newton para encontrar raiz numérica. No seguinte exemplo, nós aproximamos a raiz da função seno no intervalo [2,4]:

```
>> numeric::fsolve(sin(x), x = 2..4)
   [x = 3.141592654]
```

A função export faz funções da biblioteca "globalmente conhecida," assim que você pode usar elas sem especificar o nome da biblioteca:

```
>> export(numeric, fsolve): fsolve(sin(x), x = 2..4)
   [x = 3.141592654]
```

Se você já assinou um valor para o nome da função para ser exportada, export retorna um aviso e a função não é exportada. No seguinte, o integrador numérico numeric::quadrature não pode ser exportado para o nome quadrature porque essa identidade já tem um valor:

```
>> quadrature := 1: export(numeric, quadrature)
   Warning: 'quadrature' already has a value, not exported.
```

Depois de apagar o valor, o nome da função está disponível e a biblioteca de função correspondente pode ser exportada com sucesso. Uma pode exportar várias funções em seguida:

```
>> delete quadrature:
>> export(numeric, realroots, quadrature):
```

Agora você pode usar realroots (para encontrar *todas* as raízes reais de uma expressão em um intervalo) e quadrature (para integração numérica) diretamente. Por favor refira as páginas de ajuda correspondentes para os significados dos parâmetros de entrada e de retorno de saída.

```
>> realroots(x^4 + x^3 - 6*x^2 + 11*x - 6,
             x = -10..10, 0.001)
   [[-3.623046875, -3.62109375], [0.8217773437, 0.822265625]]
>> quadrature(exp(x) + 1, x = 0..1)
   2.718281828
```

Se você chamar export com apenas um argumento, nomeado com o nome da biblioteca, então todas as funções na biblioteca estarão exportadas. Se há conflitos de nome com identificadores juá existentes, então assuntos de export avisa:

```
>> eigenvalues := 1: export(numeric)
   Info: 'numeric::quadrature' already is exported.
   Info: 'numeric::realroots' already is exported.
   Warning: 'indets' already has a value, not exported.
   Info: 'numeric::fsolve' already is exported.
   Warning: 'rationalize' already has a value, not
            exported.
   Warning: 'linsolve' already has a value, not exported.
   Warning: 'sum' already has a value, not exported.
   Warning: 'int' already has a value, not exported.
   Warning: 'solve' already has a value, not exported.
   Warning: 'sort' already has a value, not exported.
   Warning: 'eigenvalues' already has a value, not
            exported.
```

Depois de apagar o valor do identificador eigenvalues, a biblioteca de função com o mesmo nome pode ser exportado com sucesso:

```
>> delete eigenvalues: export(numeric, eigenvalues):
```

Contudo, os outros nomes em conflito int, solved, etc. não podem ser resolvidos. A função do sistema simbólica importante int, solve etc. não devem ser recolocado pelas suas contrapartes numéricas numeric::int, numeric::solve etc.

3.3 As bibliotecas padrão

A biblioteca mais importante do MuPAD é a biblioteca padrão. Ela contém as funções mais frequentes usadas tais como diff, simplify, etc. As funções da biblioteca padrão não salvam um prefixo separado com ::, como outras funções do sistema fazem (a menos que seja exportada). Sobre este respeito, não há diferença notável entre as funções do kernel do MuPAD, as quais estão escritas em C, e as outras funções da biblioteca padrão, as quais estão implementadas na linguagem de programação do MuPAD.

Você obtém maiores informações sobre as funções disponíveis da biblioteca padrão entrando ?stdlibrary. O MuPAD Quick Reference [Oev 03] lista todas as funções da biblioteca padrão na versão do MuPAD 3.0

Muitas dessas funções são implementadas como funções ambientes (Seção 18.12). Você pode ver o código do assunto via expose (nome):

```
>> expose(exp)
   proc(x)
     name exp;
     local y, lny, c;
     option noDebug;
   begin
     if args(0) = 0 then
       error("expecting one argument")
     else
       if x::dom::exp <> FAIL then
         return(x::dom::exp(args()))
       else
         if args(0) <> 1 then
           error("expecting one argument")
         end_if
       end_if
     end_if;
     ...
   end_proc
```

Capítulo 4

Objetos do MuPAD

No capítulo 2, nós introduzimos os objetos do MuPAD tais como números, expressões simbólicas, mapas, ou matrizes. Agora, nós discutimos estes objetos mais sistematicamente.

Os objetos enviados ao kernel para avaliação podem ser de várias formas: expressões aritméticas com números tais como 1 + (1+I)/3, express~´oesx aritméticas com objetos simbólicos tais como x + (y + 1)/3, listas, conjuntos, equações, inequações, mapas, ordens, objetos matemáticos abstratos, e mais. Todo assunto do MuPAD pertence a algum tipo de dado, chamado tipo de domínio. Ele corresponde à uma certa representação interna do objeto. No que se segue, nós discutimos os seguintes tipos de domínios fundamentais. Como uma convenção, os nomes dos domínios providos pelo kernel do MuPAD consiste de cartas capitais, tais como DOM_INT, DOM_RAT Series::Puiseux ou Dom::Matrix(R) envolvendo letras pequenas:

domain type	meaning
`DOM_INT`	integers, e.g., `-3, 10^5`
`DOM_RAT`	rational numbers, e.g., `7/11`
`DOM_FLOAT`	floating-point numbers, e.g., `0.123`
`DOM_COMPLEX`	complex numbers, e.g., `1+2/3*I`
`DOM_INTERVAL`	floating-point intervals, e.g., `2.1 ... 3.2`
`DOM_IDENT`	symbolic identifiers, e.g., `x, y, f`
`DOM_EXPR`	symbolic expressions, e.g., `x+y`
`Series::Puiseux`	symbolic series expansions, e.g., `1+x+x^2+O(x^3)`
`DOM_LIST`	lists, e.g., `[1, 2, 3]`
`DOM_SET`	sets, e.g., `{1, 2, 3}`
`DOM_ARRAY`	arrays
`DOM_TABLE`	tables
`DOM_BOOL`	Boolean values: `TRUE`, `FALSE`, `UNKNOWN`
`DOM_STRING`	strings, e.g., `"I am a string"`
`Dom::Matrix(R)`	matrices and vectors over the ring `R`
`DOM_POLY`	polynomials, e.g., `poly(x^2+x+1, [x])`
`DOM_PROC`	functions and procedures

Mais ainda, você pode definir seus próprios tipos de dados, mas nós não discutimos isso aqui. A função do sistema domtype retorna o tipo domínio do objeto do MuPAD.

Na seção seguinte, nós primeiramente apresentamos a função operando importante op, possibilita a você a decompor um objeto no MuPAD dentro de seus blocos de construção. As seções seguintes discutem os tipos de dados acima e alguns dos principais sistemas de funções que os controlam.

4.1 Operando: a função op e nops

Se muitas vezes é necessário decompor um objeto do MuPAD dentro destes componentes de modo a processar eles individualmente. Os blocos de construção de um objeto são chamados *operands*. As funções do sistema para acessá-los são op e nops (abreviação para: número de operandos):

O significado de um operando depende no tipo de dado do objeto. Nós discutimos isto para cada tipo de dado em detalhe nas

```
nops(object)      : the number of operands,
op(object,i)      : the i-th operand, 0 ≤ i ≤ nops(object),
op(object,i..j)   : the sequence of operands i through j,
```

```
>> list := [a, b, c, d, sin(x)]: nops(list)
    5
>> op(list, 2)
    b
>> op(list, 3..5)
    c, d, sin(x)
>> op(list)
    a, b, c, d, sin(x)
```

seguintes seções. Por exemplo, os operandos de um número racional são o numerador e o denominador, os operandos de uma lista ou conjunto são os elementos, e os operandos de uma função dada são os argumentos. Entretanto, há também objetos onde a decomposição em operandos é menos intuitiva, tais como expansões de séries como geradas pela função do sistema Taylor ou series (Seção 4.13). Aqui é um exemplo com um conjunto (Seção 4.6):
Pela repetição chamando a função op, você pode decompor expressões arebitrárias no MuPAD dentro de umas “atômicas”. Neste modelo, um átomo do MuPAD é uma expressão que não pode ser decomposta futuramente pelo comando op, tal como op (atom)= atom controle.[2] Isso é essenc ialmente o caso de integrais, números de pontos flutuantes, identificadores que não tiveram sido assinado um valor, e cordas:

```
>> op(-2), op(0.1234), op(a), op("I am a text")
    −2, 0.1234, a, "I am a text"
```

No seguinte exemplo, a lista aninhada é decomposta completamente dentre de seus átomos a11, a12, a21, x, 2:

```
>> list := [[a11, a12], [a21, x^2]]
```

Os operandos e suboperandos são:

```
op(list, 1)                 :   [a11, a12]
op(list, 2)                 :   [a21, x^2]
op(op(list, 1), 1)          :   a11
op(op(list, 1), 2)          :   a12
op(op(list, 2), 1)          :   a21
op(op(list, 2), 2)          :   x^2
op(op(op(list, 2), 2), 1)   :   x
op(op(op(list, 2), 2), 2)   :   2
```

Ao invés das laboriosas chamadas aninhadas de op, você pode usar a seguinte forma de atalho para acessar subexpressões:

```
op(list, [1])         :   [a11, a12]
op(list, [2])         :   [a21, x^2]
op(list, [1, 1])      :   a11
op(list, [1, 2])      :   a12
op(list, [2, 1])      :   a21
op(list, [2, 2])      :   x^2
op(list, [2, 2, 1])   :   x
op(list, [2, 2, 2])   :   2
```

Exercício 4.1: Determine os operandos da potência a^b, a equação a=b, e a chamada simbólica função f (a, b).

Exercício 4.2: A seguinte chamada de solve (Capítulo 8) retorna um conjunto:

```
>> set := solve({x + sin(3)*y = exp(a),
                 y - sin(3)*y = exp(-a)}, {x,y})
```

$$\left\{\left[x = \frac{\sin(3)\,e^{-a} - e^{a} + \sin(3)\,e^{a}}{\sin(3) - 1},\ y = -\frac{e^{-a}}{\sin(3) - 1}\right]\right\}$$

Extraia os valores da solução para *y* e assine-o para a incógnita y.

4.2 Números

Nós demonstramos na seção 2.2 como trabalhar com números. Há vários tipos de dados para números:

```
>> domtype(-10), domtype(2/3), domtype(0.1234),
   domtype(0.1 + 2*I)
    DOM_INT, DOM_RAT, DOM_FLOAT, DOM_COMPLEX
```

Um num erro racional é objeto composto: os blocos de construção são o numerador e o denominador. Similarmente, um número complexo consiste da parte real e imaginária. Você pode usar a função operando op da seção anterior para acessar esses componentes:

```
>> numer(111/223), denom(111/223),
   Re(100 + 200*I), Im(100 + 200*I)
    111, 223, 100, 200
```

```
>> op(111/223, 1), op(111/223, 2)
    111, 223
>> op(100 + 200*I, 1), op(100 + 200*I, 2)
    100, 200
```

Alternativamente, você pode usar as funções do sistema numer, denom, Re, e Im:

Além dos operadores aritméticos comuns +, -, *, e /, há os operadores aritméticos div e mod para divisão de uma integral *x* por uma integral não nula *p* com resto. Se $x = k$ p + r mantém-se com integrais *k* e $0 \le r < |p|$, então x div p retorna o "quociente integral" *k* e x mod p retorna o "resto" r:

```
>> 25 div 4, 25 mod 4
    6, 1
```

`+, -, *, /, ^`	: basic arithmetic operations
`abs`	: absolute value
`ceil`	: rounding up
`div`	: quotient "modulo"
`fact`	: factorial
`float`	: approximation by floating-point numbers
`floor`	: rounding down
`frac`	: fractional part
`ifactor, factor`	: prime factorization
`isprime`	: primality test
`mod`	: remainder "modulo"
`round`	: rounding to nearest
`sign`	: sign
`sqrt`	: square root
`trunc`	: integral part

Tabela 4.2: Funções e operadores do MuPAD p-ara números

Tabela 4.2 dá uma compilação das principais funções do MuPAD e operadores para controlar números. Nós referimos a ajuda do sistema (por exemplo., ?abs, ?ceil etc.) para uma descrição detalhada destas funções. Nós tencionamos que enquanto expressões tais quais $\sqrt{2}$ matematicamente representa números, MuPAD trata eles como expressões simbólicas (Seção 4.4):

```
>> domtype(sqrt(2))
    DOM_EXPR
```

Exercício 4.3: qual é a diferença entre 1/3 + 1/3 + 1/3 e 1.0/3 + 1/3 + 1/3 no MuPAD?

Exercício 4.4: Compute as expansões decimais de π ($6^{7)}$ e $e1/3\ \pi\ \overline{\sqrt{163}}$ com a precisão de 10 e 100 dígitos, respectivamente. Qual é o 234º dígito após o ponto decimal de π?

Exercício 4.5: Após você executar x:=10^50/3.0, apenas o primeiro DIGITS dígitos decimais de x são garantidos a serem corretos.

a) Truncando a parte fracionada via trunc é então questionável. O que o MuPAD faz?

b) O que retorna para x após acrescentar DIGITS?

4.3 Identificadores

Identificadores são nomes tais como x de f que podem representar variáveis e incógnitas. Eles podem ser constituídos de combinações arbitrárias de letras, dígitos e sublinhado "_" com a única exceção que o primeiro símbolo não deve ser um dígito. MuPAD distingue o caso acima e as letras do caso abaixo. Exemplos de identificadores admissíveis são x, _x23, e the_MuPAD_system, enquanto MuPAD não vai aceitar 12x, p-2, e x>y como identificadores. MuPAD também aceita qualquer sequência de caracteres começando e terminando com uma 'marca de volta' como um identificador, então 'x>y 'é, de fato, um identificador. Nós não vamos usar tais identificadores neste tutorial.

Identificadores que não tenham assinado um valor avaliado para seu nome. No MuPAD, eles representam objetos simbólicos tais como incógnitas nas equações. Seu tipo de domínio é DOM_IDENT:

```
>> domtype(x)
    DOM_IDENT
```

Você pode nomear um objeto arbitrário para um identificador pelos sentidos do operador de alegação :=. Depois, este objeto é o *valor* do identificador. Por exemplo, depois do comando.

```
>> x := 1 + I:
```

O identificador x tem o valor 1 + I, o qual é um número complexo do tipo de domínio DOM_COMPLEX. Você deve ser cuidadoso para disatinguir entre um identificador, seu valor, e sua avaliação. Nós nos referimos importantemente no Capítulo 5, onde a estratégia de avaliação do MuPAD é descrita.

Se um identificador já tem um valor nomeado, outra tarefa escreve elaboradamente o valor prévio. A declaração y:=x não nomeia o identificador x para o identifica dor y, mas o valor corrente (a avaliação) de x:

```
>> x := 1: y := x: x, y
    1, 1
```

Se o valor de x é mudado depois de, isto não faz com que afete y:

```
>> x := 2: x, y
    2, 1
```

Contudo, se x é um identificador simbólico, o qual avalia para si mesmo, o novo identificador y refere-se a este símbolo:

```
>> delete x: y := x: x, y; x := 2: x, y
    x, x

    2, 2
```

Aqui nós deletamos o valor do identificador x pelos sentidos da função delete. Após o apagamento, x é novamente um identificador simbólico com um valor.

O operador tarefa := é uma forma abreviada da função do sistema _assign, o qual pode também ser chamada diretamente:

```
>> _assign(x, value): x
    value
```

Esta função retorna seu segundo argumento, chamado o lado da mão direita de uma tarefa. Isto explica a saída da tela após uma tarefa:

```
>> y := 2*x
    2 value
```

Você pode trabalhar com o valor retornado imediatamente. Por exemplo, a construção seguinte é permitida (a tarefa deve ser posta em parêntesis):

```
>> y := cos((x := 0)): x, y
    0, 1
```

Aqui o valor 0 é atarefado para o identificador x. O valor retornado da tarefa, por exemplo, 0, é alimentado diretamente como argumento para a função coseno, e o resultado cos (0) = 1 é nomeado para y. Assim, temos simultaneamente nomeado valores para ambos x e y.

Outra função tarefa no MuPAD é assign. Sua entrada são conjuntros ou listas de equações, as quais são transformadas dentro de tarefas:

```
>> delete x, y: assign({x = 0, y = 1}): x, y
    0, 1
```

Esta função é particularmente útil em conexão com solve (Seção 8), a qual retorna soluções como um conjunto contendo listas de equações da forma identifier = value sem nomear estes valores.

Há vários identificadores no MuPAD com valores pré-definidos. Eles representam funções matemáticas (tais como sin, exp, ouj sqrt), constantes matemáticas (tais como PI), ou algoritmo’s do MuPAD (tais como diff, int, ou limit). Se você tentar mudar o valor de um tal identificador pré definido, assuntos do MuPAD mostra um aviso de uma mensagem de erro:

```
>> sin := 1
   Error: Identifier 'sin' is protected [_assign]
```

Você pode proteger seus próprios identificadores contra reescritas via comando protect (identifier). A proteção de escrita de ambos seus próprios e dos identificadores do sistema podem ser removidos por unprotect (identifier). Entretanto, nós recomendamos fortemente para não escrever identificadores pré definidos desde que eles sejam usados por muitas funções do sistema o qual retorna m resultados imprevisíveis como redefinição. O comando anames(All) lista todos os identificadores correntes definidos.

Você pode usar o operador concatenação “.” para gerar nomes de identificadores dinamicamente. Se x e i são identificadores, então x.i gera um novo identificador pelas *avaliações* (veja capítulo 5) de x e i:

```
>> x := z: i := 2:  x.i
    z2
>> x.i := value: z2
    value
```

No exemplo seguinte, nós usamos um for loop (Capítulo 16) para nomear valores aos identificadores x1,...,x1000:

```
>> delete x:
>> for i from 1 to 1000 do x.i := i^2 end_for:
```

Devido ao lado da possibilidade efetuar ou conflitar com os identificadores já existentes, nós recomendamos fortemente usar este conceito apenas interativamente e não sem procedimentos no MuPAD.

A função do sistema genident gera um novo identificador que não foi ainda usado entes na atual sessão do MuPAD:

```
>> X3 := (X2 := (X1 := 0)): genident()
    X4
```

Você pode usar cordas inclusas nas marcas de cotação “ (Seção 4.11) para gerar identificadores dinamicamente:

```
>> a := email: b := "4you": a.b
    email4you
```

Mesmo se a corda contém brancos ou símbolos de operador, um identificador válido é gerado, o qual MuPAD mostra usando a notação remarcada mencionada acima:

```
>> a := email: b := "4you + x": a.b
    'email4you + x'
```

Cordas não são identificadores e não podem ter um valor denominado:

```
>> "string" := 1
   Error: Invalid left-hand side in assignment [line 1, \
   col 11]
```

Exercício 4.6: Quais desses seguintes nomes x, x2, 2x, x_t, diff, exp, caution!-!, x-y, Jack&Jill, a_valid_identifier são identificadores válidos no MuPAD? Quais deles podem ter valores nomeados?

Exercício 4.7: Leia a página de ajuda para solve. Resolva o sistema de equações

$x_1 + x_2 = 1$, $x_2 + x_3 = 1$,..., $x_{19} + x_{20} = 1$, $x_{20} = \pi$

nas incógnitas x_1, x_2,...,x_{20}. Leia a página de ajuda para assign e nomeie os valores da solução para as incógnitas.

4.4 Expressões simbólicas

Nós dizemos que um objeto contendo termos simbólicos tais como a equação

$$0.3 + \sin(3) + \frac{f(x,y)}{5} = 0$$

É uma *expressão*. Expressões do tipo de domínio DOM_EXPR provavelmente são o tipo mais geral de dado no MuPAD. Expressões são construídas de componentes atômicos, como todos os objetos do MuPAD, e são compostos por significados de *operadores*. Isto cumpre operadores binários, tais como as operações aritméticas básicas +, -, *, /, ^, e chamadas de funções tais como sin (.), f (.), etc.

4.4.1 Operadores

MuPAD em seu meio usa funções para combinar ou manipular objetos. Isso iria ser um pouco intuitivo, no entanto, para usar chamadas de funções em qualquer lugar, diga, _plus (a,b) para a adição *a + b*. Por esta razão, uma\ variedade de operações importantes é implementada em um caminho tal que você pode usar as notações matemáticas familiares ("operações de notação") para entrada. Também a saída é dada em uma tal forma. No seguinte, nós listamos os operadores para construir expressões mais complexas para átomos.

Os operadores +, -, *, / para as operações aritméticas básicas e ^ para exponenciação são válidos para expressões simbólicas também:

```
>> a + b + c, a - b, -a, a*b*c, a/b, a^b
    a + b + c, a - b, -a, a b c, a/b, a^b
```

Você pode colocar na entrada estes operadores no modo matemático familiar, mas internamente eles são chamadas de funções:

```
>> _plus(a,b,c), _subtract(a,b), _negate(a),
   _mult(a,b,c), _divide(a,b), _power(a,b)
    a + b + c, a - b, -a, a b c, a/b, a^b
```

O mesmo mantém para o fatorial de uma integral não negativa. Você pode colocar isso na notação matemática n!. Internamente isso é convertido para uma chamada da função fact:

```
>> n! = fact(n), fact(10)
    n! = n!, 3628800
```

Os operadores aritméticos div e mod foram apresentados no Capítulo 4.2. Eles podem também ser usados em um contexto simbólico, mas então retorna somente resultados simbólicos:

```
>> x div 4, 25 mod p
    x div 4, 25 mod p
```

Muitos objetos do MuPAD são separados por vírgulas forma uma sequência:

```
>> sequence := a, b, c + d
    a, b, c + d
```

O operador $ é uma importante ferramenta para gerar tais sequências:

```
>> i^2 $ i = 2..7 ;  x^i $ i = 1..5
```
$$4,\ 9,\ 16,\ 25,\ 36,\ 49$$

$$x,\ x^2,\ x^3,\ x^4,\ x^5$$

Equações e inequações são objetos válidos do MuPAD. Elas são geradas pelo símbolo de igualdade = e por <>, respectivamente:

```
>> equation := x + y = 2; inequality := x <> y
```
$$x + y = 2$$

$$x \neq y$$

Os operadores <, <=, >, e >= comparam as magnitudes de seus argumentos. As expressões correspondentes representam condições:

```
>> condition := i <= 2
```
$$i \leq 2$$

Em um contexto concreto, eles normalmente podem ser avaliados para um dos verdadeiros (“Booleanos”) valores TRUE ou FALSE. Tipicamente, eles são usados em declarações if ou como condições de terminação em voltas. Você pode combinar expressões Boolianas via os operadores lógicos and e or, ou negá-los via not:

```
>> condition3 := condition1 and (not condition2)
```
$$\text{condition1} \wedge \neg \text{condition2}$$

Você pode definir mapas (funções) de vários modos no MuPAD. O método mais simples é usar o operador seta -> (o símbolo de menos seguido pelo símbolo de “maior que”):

```
>> f := x -> x^2
```
$$x \mapsto x^2$$

Depois de definida uma função neste modo, você pode chamá-la como um sistema de função:

```
>> f(4), f(x + 1), f(y)
```
$$16,\ (x+1)^2,\ y^2$$

A composição de funções é definida pelos sentidos do *operador composição* @:

```
>> c := a@b: c(x)
```
$$a\,(b\,(x))$$

A repetição do operador @@ denota composição repetida de uma função com ela mesma:

```
>> f := g@@4: f(x)
```
$$g\,(g\,(g\,(g\,(x))))$$

Alguns sistemas de funções, tais como integração definida via int ou o operador $, requer uma *gama*. Você gera uma gama pelos sentidos do operador ..:

```
>> range := 0..1;  int(x, x = range)
```
$$0..1$$

$$\frac{1}{2}$$

Gamas não devem ser confundidas com intervalos de pontos flutuantes do tipo de domínio DOM_INTERVAL pelo qual o caminho pode ser criado pelo operador ... ou a função hull:

```
>> PI ... 20/3, hull(PI, 20/3)
```
$$3.141592653 \ldots 6.666666667,\ 3.141592653 \ldots 6.666666667$$

Este tipo de dado é explicado na Seção 4.17.

MuPAD trata qualquer expressão da forma identifier (argument) como um a chamada de função:

```
>> delete f:
   expression := sin(x) + f(x, y) + int(g(x), x = 0..1)
```
$$\sin(x) + \int_0^1 g\,(x)\,\mathrm{d}x + f\,(x, y)$$

Tabela 4.3 lista todos os operadores apresentados acima juntos com sua forma funcional equivalente. Você pode usar qualquer forma para introduzir expressões:

operator	system function	meaning	example
+	_plus	addition	SUM:=a+b
-	_subtract	subtraction	Difference:=a-b
*	_mult	multiplication	Product:=a*b
/	_divide	division	Quotient:=a/b
^	_power	exponentiation	Power:=a^b
div	_div	quotient modulo p	Quotient:=a div p
mod	_mod	remainder modulo p	Remainder:=a mod p
!	fact	factorial	n!
$	_seqgen	sequence generation	Sequence:=i^2$i=3..5
,	_exprseq	sequence concatenation	Seq:=Seq1,Seq2
union	_union	set union	S:=Set1 union Set2
intersect	_intersect	set intersection	S:=Set1 intersect Set2
minus	_minus	set difference	S := Set1 minus Set2
=	_equal	equation	Equation:=x+y=2
<>	_unequal	inequality	Condition:=x<>y
<	_less	comparison	Condition:=a<b
>		comparison	Condition:=a>b
<=	_leequal	comparison	Condition:=a<=b
>=		comparison	Condition:=a>=b
not	_not	negation	Condition2:=not Condition1
and	_and	logical 'and'	Condition:=a<b and b<c
or	_or	logical 'or'	Condition:=a<b or b<c
->		mapping	Square:=x -> x^2
'	D	differential operator	f'(x)
@	_fconcat	composition	h:=f@g
@@	_fnest	iteration	g:=f@@3
.	_concat	concatenation	NewName:=Name1.Name2
..	_range	range	Range:=a..b
...	interval	interval	IV:=2.1...3.5
name()		function call	sin(1),f(x),reset()

Tabela 4.3: Os principais operadores para gerar expressões no MuPAD

```
>> 2/14 = _divide(2, 14);
```
$$\frac{1}{7} = \frac{1}{7}$$
```
>> [i $ i = 3..5] = [_seqgen(i, i, 3..5)];
```
$$[3,4,5] = [3,4,5]$$
```
>> a < b = _less(a, b);
```
$$(a < b) = (a < b)$$
```
>> (f@g)(x) = _fconcat(f, g)(x)
```
$$f(g(x)) = f(g(x))$$

Nós remarcamos que algumas das funções do sistema tais como _plus, _mult, _union, ou _concat aceitam muitos argumentos arbitrariamente, enquanto os operadores correspondentes são somente binários:

```
>> _plus(a, b, u, v), _concat(a, b, u, v), _union()
```
$$a+b+u+v,\ \text{abuv},\ \emptyset$$

Isto é muitas vezes útil para conhecer e usar a forma funcional dos operadores. Por exemplo, é muito eficiente formar somas extensas pela aplicação do comando _plus para muitos argumentos. Você pode gerar a sequência de argumento pelos sentidos do gerador de sequência $:

```
>> _plus(1/i! $ i = 0..100): float(%)
```
$$2.718281828$$

A função map é uma ferramenta útil. Ela aplica uma função para todos os operandos de um objeto do MuPAD. Por exemplo, a chamada

```
>> map([x1, x2, x3], function, y, z)
```

retorna a seguinte lista (Seção 4.6):

$$[\text{function}(\text{x1}, y, z), \text{function}(\text{x2}, y, z), \text{function}(\text{x3}, y, z)]$$

Se você quer aplicar operadores via map, use seu equivalente funcional:

```
>> map([x1, x2, x3], _power, 5), map([f, g], _fnest, 5)
```
$$\left[\text{x1}^5, \text{x2}^5, \text{x3}^5\right],\ [f \circ f \circ f \circ f \circ f, g \circ g \circ g \circ g \circ g]$$

Para os operadores mais comuns, chamados +, -, *, /, ^, =,<>, <, >, <=, >=, e ==>, as funções correspondentes podem ser acessadas na forma '+', '-', '*' etc.:

```
>> map([1, 2, 3, 4], '*', 3), map([1, 2, 3, 4], '^', 2)
```
$$[3,6,9,12],\ [1,4,9,16]$$

Note que '-' é negação, não subtração.

Algumas operações são invalidas porque elas não fazem sentido matematicamente:

```
>> 3 and x
   Error: Illegal operand [_and]
```

A função do sistema _and reconhece que o argumento 3 não pode representar um valor Booleano e mostra uma mensagem de erro. Entretanto, MuPAD aceita uma expressão simbólica talo como a and b com identificadores simbólicos a, b. Logo como a e b conseguem valores Booleanos, a expressão pode ser avaliada para um valor Booleano também:

```
>> c := a and b: a := TRUE: b := TRUE: c
    TRUE
```

Os operadores tem diferentes *prioridades*, por exemplo:

```
a.fact(3) means a.(fact(3)) and returns a6,
a.6^2     means (a.6)^2 and returns a6^2,
a*b^c     means a*(b^c),
a+b*c     means a+(b*c),
a+b mod c means (a+b) mod c,
a=b mod c means a=(b mod c),
a,b$3     means a,(b$3) and returns a,b,b,b.
```

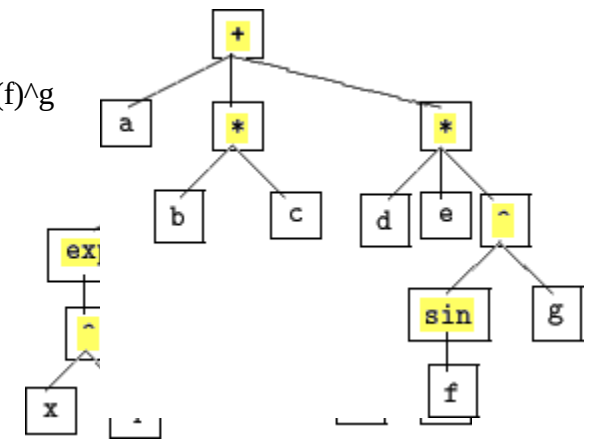
Se nós denotamos a relação “é da menor prioridade que” por <, então nós temos:
, < $ < = < mod < + < * < ^ < . < chamadas de função
Você encontra uma lista completa dos operadores e suas prioridades na Seção “Operadores” do guia rápido de referência do MuPAD. Parêntesis sempre podem ser usados para fazer cumprir uma ordem de avaliação que difere da prioridade padrão dos operandores:

```
>> 1 + 1 mod 2,  1 + (1 mod 2)
    0, 2
>> i := 2: x.i^2, x.(i^2)
    x2^2, x4
>> u, v $ 3 ;  (u, v) $ 3
    u, v, v, v

    u, v, u, v, u, v
```

4.4.2 Árvores de expressão

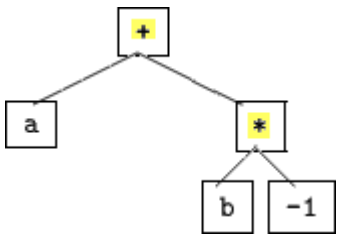
Um modelo útil para representar uma expressão no MuPAD é a *árvore de expressão*. Ela reflete a representação interna. Os operadores de suas funções correspondentes, respectivamente, são os vértices, e os seus argumentos são subárvores. O operador da mais baixa prioridade está na raiz. Aqui são alguns exemplos:

A + b * c + d * e * sin (f)^g

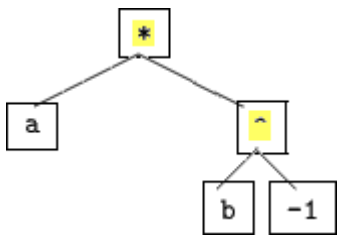


int (exp (x^4), x=0..1)

A diferença a – b é representada internamente como a + b * (-1):



Similarmente, um quociente a/b tem a representação interna a * b(-1):



As folhas de uma árvore de expressão são átomos do MuPAD.

A função prog::exprtree permite mostrar árvores de expressão. Operadores são substituídos pelos nomes das funções correspondentes do sistema:

```
>> prog::exprtree(a/b):
                        _mult
                        |
                        +-- a
                        |
                        `-- _power
                            |
                            +-- b
                            |
                            `-- -1
```

Exercício 4.8: Esboce a árvore de expressão de a^b – sin (a/b).

Exercício 4.9: determine os operandos de 2/3, x/3, 1 + 2*I, e x + 2*I. Explique a diferença que você observa.

4.4.3 Operandos

Você pode decompor expressões sistematicamente pelos sentidos das funções operando op e nops, as quais foram apresentadas na Seção 4.1. Os operandos de uma expressão correspondem às subárvores abaixo a raiz em uma árvore de expressão.

```
>> expression := a + b + c + sin(x): nops(expression)
    4
>> op(expression)
    a, b, c, sin (x)
```

Adicionalmente, expressões do tipo de domínio DOM_EXPR tem um “0º operando,” o qual é acessível via op (. ,0). Isso corresponde a raiz da árvore de expressão e diz a você qual função é usada para construir a expressão:

```
>> op(a + b*c, 0), op(a*b^c, 0), op(a^(b*c), 0),
   op(f(sin(x)),0)
    _plus, _mult, _power, f
>> sequence := a, b, c:  op(sequence, 0)
    _exprseq
```

Se a expressão é uma chamada de função simbólica, op (. , 0) retorna o identificador dessa função:

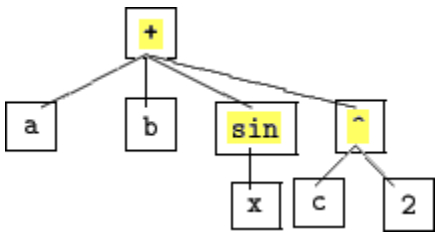
```
>> op(sin(1), 0), op(f(x), 0), op(diff(y(x), x), 0),
   op(int(exp(x^4), x), 0)
    sin, f, diff, int
```

Você pode considerar o 0º operando de uma expressão como um “tipo matemático”. Por exemplo, um algoritmo para diferenciação de expressões arbitrárias deve descobrir se a expressão é uma soma, um produto, ou uma chamada de função. Para este fim. Deve se olhar no 0º operando para decidir entre linearidade, a regra do produto, ou a regra de cadeia da diferenciação aplicada.

Como um exemplo, nós decompomos a expressão

```
>> expression := a + b + sin(x) + c^2:
```

com a árvore de expressão



Sistematicamente pelos significados da função op:

```
>> op(expression, 0..nops(expression))
```
$$_plus,\ a,\ b,\ \sin(x),\ c^2$$

Nós podemos por essas expressões juntas novamente na seguinte forma:

```
>> root := op(expression, 0): operands := op(expression):
>> root(operands)
```
$$c^2 + a + b + \sin(x)$$

No exemplo seguinte, nós decompomos uma expressão completamente dentro de seus átomos (compare com Seção 4.1):

```
>> expression := sin(x + cos(a*b)):
```

Os operandos e subexpressões são:

```
op(expression, 0)            :  sin
op(expression, 1)            :  x+cos(a*b)
op(expression, [1, 0])       :  _plus
op(expression, [1, 1])       :  x
op(expression, [1, 2])       :  cos(a*b)
op(expression, [1, 2, 0])    :  cos
op(expression, [1, 2, 1])    :  a*b
op(expression, [1, 2, 1, 0]) :  _mult
op(expression, [1, 2, 1, 1]) :  a
op(expression, [1, 2, 1, 2]) :  b
```

Exercício 4.10: Esboce a árvore de expressão da seguinte exprfessão Booleana:

```
>> condition := (not a) and (b or c):
```

Como você faz para usar op para escolher os identificadores simbólicos a, b, e c fora da **condição** do objeto?

4.5 Sequências

Sequências formam um dado de estrutura importante no MuPAD. Listas e conjuntos constroem-se de sequências. Como discutido na Seção 4.4, uma sequência são umas séries de objetos do MuPAD separados por vírgulas.

```
>> sequence1 := a, b, c; sequence2 := c, d, e
```
$$a,\ b,\ c$$

$$c,\ d,\ e$$

Você pode também usar a vírgula para concatenar sequências:

```
>> sequence3 := sequence1, sequence2
```
$$a,\ b,\ c,\ c,\ d,\ e$$

Sequências são expressões do MuPAD do tipo de domínio DOM_EXPR. Se *m* e *n* são integrais, a chamada **object (i) $ i = m..n** gera a sequência

object *(m), object (m+1),..., object (n):*

```
>> i^2 $ i = 2..7, x^i $ i = 1..5
```
$$4,\ 9,\ 16,\ 25,\ 36,\ 49,\ x,\ x^2,\ x^3,\ x^4,\ x^5$$

```
>> _seqgen(i^2, i, 2..7), _seqgen(x^i, i, 1..5)
```
$$4,\ 9,\ 16,\ 25,\ 36,\ 49,\ x,\ x^2,\ x^3,\ x^4,\ x^5$$

O comando **object $ n** retorna uma sequência de *n* objetos idênticos:

```
>> x^2 $ 10
```
$$x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2,\ x^2$$

Você pode também usar o gerador de sequência em conexão com a palavra chave **in**. A variável loop então corre através de todos os operandos do objeto manifestado:

```
>> f(x) $ x in [a, b, c, d]
```
$$f(a),\ f(b),\ f(c),\ f(d)$$
```
>> f(x) $ x in a + b + c + d + sin(sqrt(2))
```
$$f(a),\ f(b),\ f(c),\ f(d),\ f\left(\sin\left(\sqrt{2}\right)\right)$$

É fácil deixar o MuPAD executar uma sequência de comandos pelos significados de $. No exemplo seguinte, duas designações (separadas por semicolunas) são cumpridas em cada passo. Depois de tudo, os identificadores têm os valores correspondentes:

```
>> (x.i := sin(i); y.i := x.i) $ i=1..4:
>> x1, x2, y3, y4
```
$$\sin(1),\ \sin(2),\ \sin(3),\ \sin(4)$$

Como uma simples aplicação de sequências, nós consideramos agora o diferenciador do MuPAD diff. A chamada diff (f (x), x) retorna a derivada de *f* com respeito a *x*. Altas derivadas são dadas por diff (f (x), x, x), diff (f (x), x, x, x) etc. Assim, a 10ª derivada de f (x) = sin (x^2) pode ser computada convenientemente pelos significados dói gerador de sequência:

```
>> diff(sin(x^2), x $ 10)
                2              4      2
  30240 cos(x ) - 403200 x  cos(x ) +

           8      2            2      2
    23040 x  cos(x ) - 302400 x  sin(x ) +

           6      2           10      2
    161280 x  sin(x ) - 1024 x   sin(x )
```

O objeto de “anulação” do MuPAD (Seção 4.18) pode ser considerado como uma sequência vazia. Você pode gerar ela pela chamada nulo () ou _exprseq (). O sistema automaticamente elimina este objeto das sequências:

```
>> Seq := null(): Seq := Seq, a, b, null(), c
```
$$a,\ b,\ c$$

Algumas funções do sistema tais como o comando print para saída de tela (Seção 13.1.1) retorna o objeto nulo ():

```
>> sequence := a, b, print(Hello), c
                          Hello
```
$$a,\ b,\ c$$

Você pode acessar o *i*º entrada de uma sequência por **sequence [i]**. Redefinições da forma **sequence [i]:= newvalue** também são possíveis:

```
>> F := a, b, c: F[2]
```
$$b$$
```
>> F[2] := newvalue: F
```
$$a,\ \text{newvalue},\ c$$

Alternativamente, você pode usar a função operando op (Seção 4.1) para acessar subsequências.

```
>> F := a, b, c, d, e: op(F, 2); op(F, 2..4)
```
$$b$$
$$b,\ c,\ d$$

Você pode usar delete para apagar entradas de uma sequência, assim encurtando a sequência:

```
>> F; delete F[2]: F; delete F[3]: F
```
$$a,\ b,\ c,\ d,\ e$$
$$a,\ c,\ d,\ e$$
$$a,\ c,\ e$$

O principal uso de sequências no MuPAD é a geração de listas e conjuntos e argumentos de apoio para chamadas de funções. Por exemplo, a função Max e min pode computar o máximo e o mínimo, respectivamente, de quaisquer argumentos arbitrários:

```
>> Seq := 1, 2, -1, 3, 0: max(Seq), min(Seq)
```
$$3,\ -1$$

Exercício 4.11: Nomeie os valores $x_1 = 1$, $x_2 = 2$, ..., $x_{100} = 100$ para os identificadores x_1, x_2,..., x_{100}.

Exercício 4.12: Gere a sequência

$$x_1, \underbrace{x_2, x_2}_{2}, \underbrace{x_3, x_3, x_3}_{3}, \ldots, \underbrace{x_{10}, x_{10}, \ldots, x_{10}}_{10}.$$

Exercício 4.13: Use um simples comando para gerar a soma dobrada

$$\sum_{i=1}^{10}\sum_{j=1}^{i}\frac{1}{i+j}\ .$$

Dica: a função _plus aceita arbitrariamente muitos argumentos. Gere um argumento sequência convenientemente.

4.6 Listas

Como uma lista é uma sequência ordenada de objetos arbitrários do MuPAD incluído em chaves:

```
>> list := [a, 5, sin(x)^2 + 4, [a, b, c], hello,
            3/4, 3.9087]
```

$$\left[a, 5, \sin(x)^2 + 4, [a, b, c], \text{hello}, \frac{3}{4}, 3.9087\right]$$

Uma lista pode conter listas como elementos. Isso pode também ser vazio:

```
>> list := []
```

[]

A possibilidade para gerar sequências via $ é útil para construir listas:

```
>> sequence := i $ i = 1..10 : list := [sequence]
```

$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

```
>> list := [x^i $ i = 0..12]
```

$[1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}, x^{11}, x^{12}]$

Uma lista pode ocorrer do lado esquerdo de uma alegação. Isto pode ser usado para alegar valores para vários identificadores simultaneamente:

```
>> [A, B, C] := [a, b, c]: A + B^C
```

$a + b^c$

Uma propriedade útil desta construção é que todas as alegações são interpretadas no mesmo tempo, assim você pode trocar valores:

```
>> a := 1: b:= 2: a, b;
```

1, 2

```
>> [a, b] := [b, a]: a, b
```

2, 1

A função nops retorna o número de elementos de uma lista. Vocêr pode acessar os elementos de uma lista pelos sentidos da função op: op (list) retorna a sequência de elementos, por exemplo, a sequencia que já foi usada para construir a lista incluindo colchetes []. A chamada op (list, i) retorna o *i*º elemento da lista, e op (list, i..j) extra a sequência do *i*º acima a *j*º lista elemento:

```
>> delete a, b, c: list := [a, b, sin(x), c]: op(list)
```

$a,\ b,\ \sin(x),\ c$

```
>> op(list, 2..3)
```

$b,\ \sin(x)$

O operador índice provê um modo alternativo de acessar lista de elementos:

```
>> list[1], list[2]
```

$a,\ b$

Você pode mudar uma lista de elemento por uma alegação indexada:

```
>> list := [a, b, c]: list[1] := newvalue: list
```

$[\text{newvalue}, b, c]$

Você também pode usar uma gama para acesso indexado a uma sublista:

```
>> list[2..3]
```

$[b, c]$

Escrevendo à uma sublista pode mudar o comprimento de uma lista:

```
>> list[2..3] := [d,e,f,g]: list
```

$[\text{newvalue}, d, e, f, g]$

Alternativamente, o comando subsop (list, i = newvalue) (Capítulo 6) retorna uma cópia com o *i*º operando redefinido:

```
>> list := [a, b, c]: list2 := subsop(list, 1 = newvalue)
```

$[\text{newvalue}, b, c]$

Atenção: Se L é um identificador sem um valor, então a alegação indexada

```
>> L[index] := value:
```

gera uma tabela (Seção 4.8) e não uma lista:

```
>> delete L: L[1] := a: L
```
$$\left[1 = a \right.$$

Você pode remover elementos de uma lista usando delete. Isso encurta a lista:

```
>> list := [a, b, c]: delete list[1]: list
```
$$[b, c]$$

A função contains verifica se um objeto do MuPAD pertence à uma lista. Ela retorna o índice de (a primeira ocorrência de) o elemento na lista. Se a lista não contém o elemento, então contains retorna a integral 0:

```
>> contains([x + 1, a, x + 1], x + 1)
```
$$1$$
```
>> contains([sin(a), b, c], a)
```
$$0$$

A função append adiciona elementos para seguir a fileira de uma lista:

```
>> list := [a, b, c]: append(list, 3, 4, 5)
```
$$[a, b, c, 3, 4, 5]$$

O operador ponto . concatena listas:

```
>> list1 := [1, 2, 3]: list2 := [4, 5, 6]:
>> list1.list2, list2.list1
```
$$[1, 2, 3, 4, 5, 6],\ [4, 5, 6, 1, 2, 3]$$

A função correspondente do sistema é _concat e aceita arbitrariamente vários argumentos. Você pode usar isso para combinar várias listas:

```
>> _concat(list1 $ 5)
```
$$[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]$$

Uma lista pode ser encurtada pelos sentidos da função sort. Isto arranja valores numéricos de acordo com sua magnitude, cordas (Seção 4.11) são encurtados lexicograficamente:

```
>> sort([-1.23, 4, 3, 2, 1/2])
```
$$\left[-1.23, \frac{1}{2}, 2, 3, 4\right]$$
```
>> sort(["A", "b", "a", "c", "C", "c", "B", "a1", "abc"])
```
$$[\text{"A"}, \text{"B"}, \text{"C"}, \text{"a"}, \text{"a1"}, \text{"abc"}, \text{"b"}, \text{"c"}, \text{"c"}]$$
```
>> sort(["x10002", "x10011", "x10003"])
```
$$[\text{"x10002"}, \text{"x10003"}, \text{"x10011"}]$$

Note que a ordem lexicográfica apenas aplica para cordas geradas com “. Nomes de identificadores são encurtados de acordo com diferentes (internas) regras, das quais, apesar de outras coisas, levam o comprimento dos nomes dentro da conta:

```
>> delete A, B, C, a, b, c, a1, abc:
   sort([A, b, a, c, C, c, B, a1, abc])
```
$$[A, B, C, a, \text{a1}, \text{abc}, b, c, c]$$
```
>> sort([x10002, x10011, x10003])
```
$$[\text{x10002}, \text{x10011}, \text{x10003}]$$

MuPAD considera listas de nomes de função como listas de funções avaliadas:

```
>> [sin, cos, f](x)
```
$$[\sin(x), \cos(x), f(x)]$$

A função map aplica uma função para todos os elementos de uma lista:

```
>> map([x, 1, 0, PI, 0.3], sin)
```
$$[\sin(x), \sin(1), 0, 0, 0.2955202067]$$

Se a função têm mais que um argumento, então map substitui os elementos da lista para o primeiro argumento e leva os argumentos restantes de sua própria lista de argumentos:

```
>> map([a, b, c], f, y, z)
```
$$[f(a, y, z), f(b, y, z), f(c, y, z)]$$

Esta construção map é uma ferramenta poderosa para controlar listas tão bem como outros objetos do MuPAD. No seguinte exemplo, nós temos uma lista aninhada L. Nós queremos extrair os primeiros elementos da sublista usando op (. , 1). Isto é facilmente feito usando map:

```
>> L := [[a1, b1], [a2, b2], [a3, b3]]: map(L, op, 1)
```
$$[\text{a1}, \text{a2}, \text{a3}]$$

A função do MuPAD permite à você extrair elementos com uma certa propriedade de uma lista. Para esta finalidade, você precisa de uma função que verifica se um objeto têm esta propriedade e retorna TRUE ou FALSE. Por exemplo, a chamada hás (object1, object2) retorna TRUE se object2 é um operando ou suboperando de object1, e FALSE o contrário:

```
>> has(1 + sin(1 + x), x), has(1 + sin(1 + x), y)
```
TRUE, FALSE

Agora,

```
>> select([a + 2, x, y, z, sin(a)], has, a)
```
$[a+2, \sin(a)]$

extrai todas essas listas de elementos para cada has (., a) retorna TRUE, por exemplo a estas as quais contêm o identificador **a**.

A função split divide uma lista dentro de três listas, como segue. A primeira lista contém todos os elementos com uma certa propriedade, a segunda lista coleta todos esses elementos sem a propriedade. Se o teste para propriedade retornar o valor UNKNOWN para alguns elementos, então esses são colocados dentro da terceira lista. Do lado contrário, a terceira lista está vazia. A função split retorna uma lista incluindo as três listas descritas acima:

```
>> split([sin(x), x^2, y, 11], has, x)
```
$[[\sin(x), x^2], [y, 11], []]$

A função do MuPAD zip combina elementos de duas listas emparelhadas dentro de uma nova lista:

```
>> L1 := [a, b, c]: L2 := [d, e, f]:
>> zip(L1, L2, _plus), zip(L1, L2, _mult),
   zip(L1, L2, _power)
```
$[a+d, b+e, c+f],\ [a\,d, b\,e, c\,f],\ \left[a^d, b^e, c^f\right]$

O terceiro argumento de zip deve ser uma função que leva dois argumentos. Esta função é então aplicada para os pares dos elementos da lista. No exemplo acima, nós usamos a funções do MuPAD _plus, _mult, e _power parfa adição, multiplicação, e exponenciação, respectivamente. Se as duas listas de entradas têm diferentes comprimentos, então o comportamento de zip depende do quarto argumento opcional. Se isto não for apresentado, então o comprimento da lista resultante é o mínimo de comprimento das duas listas de entradas. Do contrário, se você fornecer um quarto argumento, então zip substitui as entradas de lista “faltando” pelo seu argumento:

```
>> L1 := [a, b, c, 1, 2]: L2 := [d, e, f]:
>> zip(L1, L2, _plus)
```
$[a+d, b+e, c+f]$
```
>> zip(L1, L2, _plus, hello)
```
$[a+d, b+e, c+f, \text{hello}+1, \text{hello}+2]$

Aqui é um sumário de toda lista de operações que nós discutimos:

`.` or `_concat`	:	concatenating lists
`append`	:	appending elements
`contains(list, x)`	:	does `list` contain the element `x`?
`list[i]`	:	accessing the i-th element
`map`	:	applying a function
`nops`	:	length
`op`	:	accessing elements
`select`	:	select according to properties
`sort`	:	sorting
`split`	:	split according to properties
`subsop`	:	replacing elements
`delete`	:	deleting elements
`zip`	:	combining two lists

Exercício 4.14: Gere duas listas com os eloementos *a, b, c, d* e 1, 2, 3, 4, respectivamente. Concatene as listas. Multiplique as listas emparelhadas

Exercício 4.15: Multiplique todas as entradas da lista [1, x, 2] por 2. Suponha estar dada uma lista, quais elementos são listas de números ou expressões, tais como [[1, x, 2], [PI], [2/3, 1]], como você polde multiplicar todas as entradas por 2?

Exercício 4.16: Deixe X = [$x_{1,...,}\ x_n$] e Y = [y_1,..., y_n] serem duas listas do mesmo comprimento. Encontre um método simples para computar

- seu “produto interno” (X como vetor de fileira e Y como vetor de coluna)

$$x_1y_1 + ... + x_ny_n,$$

- seu “produto matriz” (X como vetor de coluna e Y como vetor de

fileira) [[x_1y_1, x_1y_2,...,x_1y_n], [x_2y_1, x_2y_2,…,x_2y_n],
…, [x_ny_1, x_n y_2,…, x_ny_n]]

Você pode alcançar isto pelo uso de zip, _plus, map e funções apropriadas (Seção 4.12) sem uma única linha de comando em cada caso. Loops (Capítulo 16) não são requeridos.

Exercício 4.17: na teoria dos números, um é frequentemente interessado na densidade de números primos na sequência da forma $f(1)$, $f(2)$,..., onde f é um polinômio. Para cada valor de m = 0, 1,..., 100 são primos

Exercício 4.18: Em que ordenando vai *n* crianças serem eliminadas por uma contada-se rima composta por *m* palavras? Por exemplo, usando o rima

> "eenie-meenie-miney-moe-catch a-tiger-by the toe"

Com 8 "palavras", 12 crianças são eliminadas na ordem 8-4-1-11-10-12-3-7-6-2-9-5. Dica: represente a criança por uma lista [1, 2,...] e remova um elemento dessa lista depois do término da contagem.

`contains(M, x)`	:	does `M` contain the element `x`?
`intersect`	:	intersection
`map`	:	applying a function
`minus`	:	set-theoretic difference
`nops`	:	number of elements
`op`	:	accessing elements
`select`	:	select according to properties
`split`	:	split according to properties
`subsop`	:	replacing elements
`union`	:	set-theoretic union

Tabela 4.4: Funções do MuPAD e operadores para conjuntos

4.7 Conjuntos

Um conjunto é uma sequência desordenada de objetos arbitrários incluídos em suspensórios ondulados. Conjuntos são do tipo de domínio DOM_SET:

```
>> {34, 1, 89, x, -9, 8}
   {1, 8, -9, 34, 89, x}
```

A ordem dos elementos num MuPAD parece ser aleatória. O kernel do MuPAD os elementos de acordo com princípios internos. Você deve usar conjuntos a\penas se a ordem dos elementos não importar. Se você quer processar uma sequência de expressões em certa ordem, use listas como discutido na seção anterior.

Conjuntos podem ser vazios:

```
>> emptyset := {}
   ∅
```

Um conjunto contém cada elemento apenas após, por exemplo, duplicados os elementos são removidos automaticamente:

```
>> set := {a, 1, 2, 3, 4, a, b, 1, 2, a}
   {1, 2, 3, 4, a, b}
```

A função nops determina o número de elementos em um conjunto. Como para sequências e listas, op extrai elementos de um conjunto.

```
>> op(set)
   a, 1, 2, 3, 4, b
>> op(set, 2..4)
   1, 2, 3
```

Atenção: Desde que elementos de um conjunto podem ser reordenados internamente, você deve checar cuidadosamente se isso faz sentido para acessar o *i*º elemento. Por exemplo, subsop (set, i = newvalue) (Seção 6) substitui o *i*º elemento por um novo valor. Contudo, você deve checar em avanço (usando op) que o elemento como impresso na tela.

As funções union, intersect, e minus formam a união, a intersecção, e o conjunto da diferença teoretica, respectivamente, de conjuntos:

```
>> M1 := {1, 2, 3, a, b}: M2 := {a, b, c, 4, 5}:
>> M1 union M2, M1 intersect M2, M1 minus M2, M2 minus M1
   {1, 2, 3, 4, 5, a, b, c}, {a, b}, {1, 2, 3}, {4, 5, c}
```

Em particular, você pode usar minus para remover elementos de um conjunto:

```
>> {1, 2, 3, a, b} minus {3, a}
   {1, 2, b}
```

Você pode também substituir um elemento por um novo valor sem se importar com a ordem dos elementos:

```
>> delete a, b, c, d: set := {a, b, oldvalue, c, d}
   {a, b, c, d, oldvalue}
>> set minus {oldvalue} union {newvalue}
   {a, b, c, d, newvalue}
```

A função contains verifica se um elemento pertence a um conjunto, e retorna se é TRUE ou FALSE.

```
>> contains({a, b, c}, a), contains({a, b, c + d}, c)
   TRUE, FALSE
```

MuPAD considera conjuntos nomes de função para todos os elementos de um conjunto pelos significados de map:

```
>> map({x, 1, 0,  PI, 0.3}, sin)
```
$$\{0.2955202067, 0, \sin(1), \sin(x)\}$$

Você pode usar a função select para extrair elementos com uma certa propriedade de um conjunto. Ela trabalha como para listas, mas o objeto retornado é um conjunto:

```
>> select({{a, x, b}, {a}, {x, 1}}, contains, x)
```
$$\{\{1, x\}, \{a, b, x\}\}$$

Similarmente, você pode usar a função split para dividir num conjunto dentro de três subconjuntos de elementos com uma certa propriedade, elementos sem essa propriedade, e elementos por quais o sistema não pode decidir isto e retorna UNKNOWN. O resultado é uma lista abrangendo estes três jogos:

```
>> split({{a, x, b}, {a}, {x, 1}}, contains, x)
```
$$[\{\{1, x\}, \{a, b, x\}\}, \{\{a\}\}, \emptyset]$$

Tabela 4.4 contém um sumário das operações de conjuntos discutidas tão distante.

A biblioteca combinate contém algumas funções combinatórias para jogos finitos. Chame ?combinat para obter um guia de sobrevivência. Por exemplo, este pacote contém a função combinat::subsets, a qual retorna a potência do conjunto de um dado conjunto (veja ?combinat::subsets para mais informações).

MuPAD também provê a estrutura de dado Dom::ImageSet para controlar conjuntos infinitos; ver Seção 8.2.

Exercício 4.19: Como você pode converter uma lista de um conjunto e vice versa?

Exercício 4.20: Gere os conjuntos A = {a, b, c}, B = {b, c, d}, e C = {b, c, e}. Compute as uniões e a intersecções dos três conjuntos, tão bem como a diferença A \ (B ∪ C).

Exercício 4.21: Ao invés dos operadores binários intersect e union, você pode usar também as funções correspondentes do MuPAD _intersect e _union para computar uniões e intersecções de conjuntos. Estas funções aceitam arbitrariamente muitos argumentos. Use simples comandos para computar a união e a intersecção de todos os conjuntos pertencentes a M:

```
>> M := {{2, 3}, {3, 4}, {3, 7}, {5, 3}, {1, 2, 3, 4}}:
```

Exercício 4.22: A biblioteca combinat contém uma função geradora de todos os subconjuntos *k* de um conjunto finito. Encontre essa função e leia a página de ajuda correspondente. Quantos tantos deles estão ali?

4.8 Tabelas

Uma tabela é um objeto do MuPAD do tipo de domínio DOM_TABLE. Esta é a coleção de equações da forma índex = value. Ambos os índices e valores podem ser objetos arbitrários do MuPAD. Você pode gerar uma tabela por uso da função do sistema table (“geração explicita”):

```
>> T := table(a = b, c = d)
```
$$\begin{bmatrix} a & = & b \\ c & = & d \end{bmatrix}$$

Você pode gerar mais entradas ou mudar as existentes por “índices tarefas” da forma Table [índex]: = value:

```
>> T[f(x)] := sin(x): T[1, 2] := 5:
>> T[1, 2, 3] := {a, b, c}: T[a] := B:
>> T
```
$$\begin{bmatrix} a & = & B \\ c & = & d \\ f(x) & = & \sin(x) \\ (1,2) & = & 5 \\ (1,2,3) & = & \{a, b, c\} \end{bmatrix}$$

Não é necessário inicializar uma tabela via table. Se T é um identificador que não tem um valor, então uma alegação indexada da forma T [índex]:= value automaticamente torna T dentro de uma tabela (“geração implícita”):

```
>> delete T: T[a] := b: T[b] := c: T
```
$$\begin{bmatrix} a & = & b \\ b & = & c \end{bmatrix}$$

Uma tabela pode ser vazia:

```
>> T := table()
```
$$[$$

Você pode apagar entradas via delete Table [índex]:

```
>> T := table(a = b, c = d, d = a*c):
>> delete T[a], T[c]: T
```

$$\left[d = a\,c \right.$$

Você pode acessar entradas de tabela na forma Table [índex]; isto retorna o elemento correspondente para o índice. Se não há entrada para o índice, então MuPAD retorna Table [índex] simbolicamente:

```
>> T := table(a = b, c = d, d = a*c):
>> T[a], T[b], T[c], T[d]
```

$$b,\ T_b,\ d,\ a\,c$$

A chamada op (Table) retorna todas as entradas de uma tabela, por exemplo, a sequência de todas as equações índex = value:

```
>> op(table(a = A, b = B, c = C, d = D))
```

$$a = A,\ b = B,\ c = C,\ d = D$$

Note que a ordem interna das entradas da tabela podem diferir da ordem a qual você tinha gerado a tabela. Isso pode parecer totalmente aleatório:

```
>> op(table(a.i = i^2 $ i = 1..17))
   a9 = 81, a10 = 100, a8 = 64, a11 = 121, a7 = 49,

      a12 = 144, a6 = 36, a13 = 169, a5 = 25, a14 = 196,

      a4 = 16, a15 = 225, a3 = 9, a16 = 256, a2 = 4,

      a17 = 289, a1 = 1
```

A função map aplica uma dada função para os *valores* (não os *índices*) de todas as entradas da tabela:

```
>> T := table(1 = PI, 2 = 4, 3 = exp(1)): map(T, float)
```

$$\left[\begin{array}{l} 1 = 3.141592654 \\ 2 = 4.0 \\ 3 = 2.718281828 \end{array} \right.$$

A função contains checa se uma tabela contém um *índice* particular. Isso ignora os *valores:*

```
>> T := table(a = b): contains(T, a) , contains(T, b)
```

TRUE, FALSE

Você pode empregar as funções select e split para inspecionar *ambos índices e valores* de uma tabela e para extrair eles de acordo com certas propriedades. Isto trabalha de modo similar como para listas (Seção 4.6) e para conjuntos (Seção 4.7):

```
>> T := table(1 = "number", 1.0 = "number", x = "symbol");
>> select(T, has, "symbol")
```

$$\left[x = \text{"symbol"} \right.$$

```
>> select(T, has, 1.0)
```

$$\left[1.0 = \text{"number"} \right.$$

```
>> split(T, has, "number")
```

$$\left[\left[\begin{array}{l} 1 = \text{"number"} \\ 1.0 = \text{"number"} \end{array} \right. , \left[x = \text{"symbol"} \right. , [\right]$$

Tabelas são particularmente bem vestidas para armazenamento de grandes quantidades de dados. Acessos indexados para elementos *individuais* são implementados eficientemente também para grandes tabelas: uma escrita ou lida não armazena através de toda estrutura de dado.

Exercício 4.23: Gere uma tabela **telephoneDirectory** com as seguintes entradas:

Ford 1815, Reagan 4711, Bush 1234, Clinton 5678

Olhe para o número de Ford. Como você pode encontrar afora de quem é 5678?

Exercício 4.24: Dada uma tabela, como você pode gerar uma lista de todas as entradas de todos os índices e uma lista de todos os valores, respectivamente?

Exercício 4.25: Gere a tabela **table** (1 = 1, 2 = 2,..., n = n) e a lista [[1, 2,..., n] do comprimento *n* = 100000. Adicione uma nova entrada para a tabela e para a lista. Quão longa isto toma? Dica: a chamada time ((a:=b)) retorna o tempo de execução de uma alegação.

4.9 Arranjos

Arranjos, do tipo de domínio DOM_ARRAY, podem ser considerados como tabelas especiais. Você pode pensar delas como uma coleção de equações da forma
índex = value, mas em contraste com tabelas, os índices devem ser integrais. Um arranjo uni dimensional consiste de equações da forma **i = value.** Matematicamente, isso representa um vetor pelo qual o *i*º componente é **value.** Uma ordem duo dimensional representa uma matriz, pela qual os componentes (i, j)º estão armazenados na forma (i, j)= **value**. Você pode gerar arranjos de dimensões arbitrárias, com entradas da forma (i, j, k,...) = **value.**

A função do sistema array gera arranjos. Em sua mais simples forma, você apenas especifica uma sequência de gamas que determinam a dimensão e o tamanho do arranjo:

```
>> A := array(0..1, 1..3)
                +-                                -+
                |  ?[0, 1], ?[0, 2], ?[0, 3]  |
                |                                |
                |  ?[1, 1], ?[1, 2], ?[1, 3]  |
                +-                                -+
```

Você pode ver aqui que a primeira gama 0..1 e a segunda gama 1..3 determina o número da gama de fileiras e colunas, respectivamente. A saída ? [0, 1] sinaliza que o índice correspondente não tem ainda um valor nomeado. Assim, o comando acima gerou um arranjo vazio. Agora, você pode designar valores para os índices:

```
>> A[0, 1] := 1: A[0, 2] := 2: A[0, 3] := 3:
>> A[1, 3] := HELLO: A
```

$$\begin{pmatrix} 1 & 2 & 3 \\ A_{1,1} & A_{1,2} & \text{HELLO} \end{pmatrix}$$

Você pode também inicializar o arranjo completo diretamente quando gera ele por array. Apenas forneça os valores como uma lista (aninhada):

```
>> A := array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Você pode acessar e modificar elementos de arranjos da mesma forma como elementos de tabela:

```
>> A[2, 3] := A[2, 3] + 10: A
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 16 \end{pmatrix}$$

Novamente, você pode apagar um elemento de arranjo pelo uso de delete:

```
>> delete A[1, 1], A[2, 3]:  A , A[2, 3]
```

$$\begin{pmatrix} A_{1,1} & 2 & 3 \\ 4 & 5 & A_{2,3} \end{pmatrix}, A_{2,3}$$

Arranjos possuem um “operando 0º” **op (Array, 0)** provendo informação na dimensão e o tamanho de um arranjo. A chamada op (Array, 0) retorna uma sequência d, $a_1..b_1$, ... , $a_d..b_d$, onde *d* é a dimensão (por exemplo, o numero de índices) e $a_i..b_i$ é a gama válida para o iº índice:

```
>> Vector := array(1..3, [x, y, z]): op(Vector, 0)
    1, 1..3
>> Matrix := array(1..2, 1..3, [[a, b, c], [d, e, f]]):
>> op(Matrix, 0)
    2, 1..2, 1..3
```

Assim, a dimensão de uma matriz *m* X *n* **array** (1..m, 1..n) pode ser obtido como

M = op (Matrix, [0, 2, 2]), n = op (Matrix, [0, 3, 2]).

A estrutura interna de arranjos difere da estrutura de tabelas. As entradas não são guardadas na forma de equações:

```
>> op(Matrix)
    a, b, c, d, e, f
```

O tipo de tabela é mais flexível que o tipo de arranjo: tabelas admitem índices arbitrários, e seu tamanho pode crescer dinamicamente. Arranjos são pretendidos para armazenar vetores e matrizes de um tamanho fixo. Quando você entra com uma chamada indexada, o sistema checa se os índices estão dentro das gamas específicas. Por exemplo:

```
>> Matrix[4, 7]
   Error: Illegal argument [array]
```

Você pode aplicar uma função para todos os componentes do arranjo via map. Por exemplo, aqui está o meio mais simples para converter todas as entradas de arranjos para números de pontos flutuantes:

```
>> A := array(1..2, [PI, 1/7]): map(A, float)
```

$$\begin{pmatrix} 3.141592654 & 0.1428571429 \end{pmatrix}$$

Atenção: Se m é um identificador sem um valor, então uma alegação indexada da forma M [índex, índex, ...]: = value gera uma tabela e não um arranjo do tipo DOM_ARRAY (Seção 4.8):

```
>> delete M: M[1, 1] := a: M
```

$$\begin{bmatrix} (1,1) & = & a \end{bmatrix}$$

Adicionalmente, MuPAD provê as mais poderosas estruturas de dados de tipo de domínio Dom::Matrix para manipular vetores e matrizes. Estas são discutidas na Seção 4.15. Tais objetos são muito convenientes para se usar: você pode multiplicar duas matrizes ou uma matriz e um vetor pelos sentidos do usual símbolo de multiplicação *. Similarmente, você pode adicionar matrizes de igual dimensão via +. Para alcance da mesma funcionalidade com arranjos, você têm que escrever seus próprios procedimentos. Nós nos referimos aos exemplos MatrixProduct e MatrixMult nas Seções 18.4 e 18.5, respectivamente.

Exercício 4.26: Gere uma assim chamada matriz de Hilbert H de dimensão 20 X 20 com entradas $H_{ij} = 1/(i – j – 1)$.

4.10 Expressões Booleanas

MuPAD implementa três valores lógicos (“Booleanos”): TRUE, FALSE, e UNKNOWN:

```
>> domtype(TRUE), domtype(FALSE), domtype(UNKNOWN)
    DOM_BOOL, DOM_BOOL, DOM_BOOL
```

Os operadores and, or, e not operam em valores Booleanos:

```
>> TRUE and FALSE, not (TRUE or FALSE), TRUE and UNKNOWN,
   TRUE or UNKNOWN
    FALSE, FALSE, UNKNOWN, TRUE
```

A função bool avalia equações, inequações, ou comparações via >, >=, <, <=, para TRUE ou FALSE:

```
>> a := 1: b := 2:
>> bool(a = b), bool(a <> b),
   bool(a <= b) or not bool(a > b)
    FALSE, TRUE, TRUE
```

Note que bool somente pode ser usado para comparar números reais no MuPAD do tipo de domínio DOM_INT (integrais), DOM_RAT (números racionais) ou DOM_FLOAT (números de real pontos-flutuantes), respectivamente. Expressões numéricas exatas tais como sqrt (2), exp (3) ou PI não podem ser comparadas.

```
>> bool(3 <= PI)
   Error: Can’t evaluate to boolean [_leequal]
```

Tipicamente, você vai usar construções Booleanas em condições ramificadas de instruções if (Capítulo 17) ou em condições de terminação de loops repeat (Capítulo 16). No seguinte exemplo, nós testamos as integrais 1, 2, 3 para primária. A função do sistema isprime (“ é o argumento um número primo?”) retorna TRUE ou FALSE. O loop repeat para assim que a condição de término i = 3 avalia para TRUE:

```
>> i := 0:
   repeat
     i := i + 1;
     if isprime(i)
        then print(i, "is a prime")
        else print(i, "is no prime")
     end_if
   until i = 3 end_repeat
                     1, "is no prime"

                     2, "is a prime"

                     3, "is a prime"
```

Aqui nós usamos cordas incluídas em “ para a saída da tela. Elas são discutidas em detalhes na Seção 4.11. Note que não é necessário usar a função bool em ramificação ou condições de terminação na ordem para avaliar a condição para TRUE ou FALSE.

Exercício 4.27: Deixe **Λ** denotar a lógica “and”, deixe ∨ denotar a lógica “or”, deixe ¬ denotar a negação lógica. Para cada um dos valores Booleanos faz

TRUE Λ (FALSE ∨ ¬ FALSE ∨ ¬ FALSE))

Avalie?

Exercício 4.28: Deixe L1, L2 serem duas listas do MuPAD de comprimentos iguais. Como você encontra se L1 [i] < l2 [i] mantém verdade para todas as listas de elementos?

4.11 Fileiras

Fileiras são peças de texto, as quais podem ser usadas para saída de telas formatada. Uma corda é uma sequência de símbolos arbitrários incluídos em “delimitadores de corda” “. Este tipo de domínio é DOM_STRING.

```
>> string1 := "Use * for multiplication";
   string2 := ", ";
   string3 := "use ^ for exponentiation."
                "Use * for multiplication"

                           ", "

              "use ^ for exponentiation."
```

O operador de concatenação . combina fileiras:

```
>> string4 := string1.string2.string3
   "Use * for multiplication, use ^ for exponentiation."
```

O operador ponto é uma forma abreviada da função do MuPAD _concat, que concatena (arbitrariamente tantas)fileiras:

```
>> _concat("This is ", "a string", ".")
                   "This is a string."
```

O operador índice [] extrai os caracteres de uma corda:

```
>> string4[1], string4[2], string4[3],
   string4[4], string4[5]
                  "U", "s", "e", " ", "*"
```

Você pode usar o comando print para sair resultados intermediários nos loops ou procedimentos na tela (Seção 13.1.1). Por padrão, esta função imprime fileiras com a inclusão de duplas citações. Você pode mudar este comportamento usando a opção **Unquoted:**

```
>> print(string4)
   "Use * for multiplication, use ^ for exponentiation."

>> print(Unquoted, string4)
    Use * for multiplication, use ^ for exponentiation.
```

Fileiras não são identificadores válidos no MuPAD, assim você não pode denominar valores à elas:

```
>> "name" := sin(x)
   Error: Invalid left-hand side in assignment [line 1, \
   col 9]
```

Também, aritmética com fileiras não é permitida:

```
>> 1 + "x"
   Error: Illegal operand [_plus]
```

Contudo, você pode usar fileiras em equações:

```
>> "derivative of sin(x)" = cos(x)
    "derivative of sin(x)" = cos(x)
```

A função expr2text converte um objeto do MuPAD à uma fileira. Você pode empregar esta função para customizar comandos de impressão:

```
>> i := 7:
>> print(Unquoted, expr2text(i)." is a prime.")
                       7 is a prime.

>> a := sin(x):
>> print(Unquoted, "The derivative of " . expr2text(a) .
                   " is " . expr2text(diff(a, x)). ".")
            The derivative of sin(x) is cos(x).
```

A documentação de print contém exemplos mais avançados para saídas de usuário definida, veja ?print

Você encontra numerosas outras funções úteis para controlar cordas na biblioteca padrão (Seção “Manipulação de fileiras” do guia rápido de referência do MuPAD e na biblioteca string (stringlib).

Exercício 4.29: Na Seção 4.3, nós já mencionamos o comando **anames** (All), o qual retorna um conjunto de todos os identificadores que tem um valor na atual sessão. Gere uma lista *lexicograficamente* ordenada destes identificadores.

Exercício 4.30: Como você pode obter a “imagem espelho” de uma corda? Dica: a função **lenght** retorna o número de símbolos em uma corda.

4.12 Funções

O operador seta -> (um símbolo de menos seguido do símbolo “maior que”) gera objetos simples que representam funções matemáticas:

```
>> f := (x, y) -> x^2 + y^2
    (x, y) ↦ x^2 + y^2
```

A função **f** pode agora ser chamada como qualquer sistema de função. Isso leva dois parâmetros de entrada arbitrários (ou “argumentos”) e retorna a soma de seus quadrados:

```
>> f(a, b + 1)
    (b + 1)^2 + a^2
```

No exemplo seguinte, o valor retornado da função é gerado por uma declaração **if**:

```
>> absValue := x -> (if x >= 0 then x else -x end_if):
>> absValue(-2.3)
    2.3
```

Como discutido na Seção 4.4.1, o operador @ gera a composição h : $x \rightarrow f(g(x))$ de duas funções *f* e *g*:

```
>> f := x -> 1/(1 + x): g := x -> sin(x^2):
>> h := f@g: h(a)
          1
    ------------
    sin(a^2) + 1
```

Você pode definir uma composição repetida *f (f (f (.)))* de uma função com si mesma usando o operador repetição @@:

```
>> fff := f@@3: fff(a)
```

$$\frac{1}{\frac{1}{\frac{1}{a+1}+1}+1}$$

Com certeza, estas construções também trabalham para funções de sistema. Por exemplo, a função **abs @ Re** computa o valor absoluto da parte real de um numero complexo:

```
>> f := abs@Re: f(-2 + 3*I)
```

$$2$$

Em computações simbólicas, você muitas vezes têm a escolha de representar uma função matemática ou como um mapa **arguments ↣ value** ou como uma *expressão:*

```
>> Map := x -> 2*x*cos(x^2):
>> Expression := 2*x*cos(x^2):
>> int(Map(x), x), int(Expression, x)
```

$$\sin\left(x^2\right),\ \sin\left(x^2\right)$$

Você pode facilmente converter entre esses dois gêneros de representação. Por exemplo, a função **unapply** da biblioteca **fp** converte uma expressão para uma função, a qual você pode manipular futuramente:

```
>> h := fp::unapply(Expression);
```

$$x \mapsto 2\,x\,\cos\left(x^2\right)$$

```
>> h'
```

$$x \mapsto 2\,\cos\left(x^2\right) - 4\,x^2\,\sin\left(x^2\right)$$

De fato, **h'** é o equivalente funcional de **diff (Expression, x):**

```
>> h'(x) = diff(Expression, x)
```

$$2\,\cos\left(x^2\right) - 4\,x^2\,\sin\left(x^2\right) = 2\,\cos\left(x^2\right) - 4\,x^2\,\sin\left(x^2\right)$$

MuPAD pode representar mapas pelos significados de *expressões funcionais*: funções mais complexas são construídas de simples funções (tais como **sin**, **cos**, **exp**, **ln**, **id**) pelos sentidos dos operadores (tais como o operador composição @ ou os operadores aritméticos +, *, etc.). Note que os operadores aritméticos geram funções que são definidas *ponto sensato*, o qual é matematicamente sonoro. Por exemplo, *f + g* representa o mapa $x \supset f(x) + g(x)$, *f. g* representa o mapa $x \supset f(x).\ g(x)$, etc.:

```
>> delete f, g:
>> a := f + g: b := f*g: c := f/g: a(x), b(x), c(x)
```

$$f(x) + g(x),\ f(x)\ g(x),\ \frac{f(x)}{g(x)}$$

Você tem a permissão de ter valores numéricos em expressões funcionais. MuPAD considera eles como funções constantes as quais sempre retornam o valor particular:

```
>> 1(x), 0.1(x, y, z), PI(x)
```

$$1,\ 0.1,\ \pi$$

```
>> a := f + 1: b := f*3/4: c := f + 0.1: d := f + sqrt(2):
>> a(x), b(x), c(x), d(x)
```

$$f(x) + 1,\ \frac{3\,f(x)}{4},\ f(x) + 0.1,\ f(x) + \sqrt{2}$$

O operador -> é útil para definir funções que retornam valores que podem ser obtidos por simples operações. Funções implementando algoritmos mais complexos normalmente requerem muitos comandos e variáveis auxiliares para armazenar resultados intermediários. Em princípio, você pode definir tais funções via -> também. Contudo, isto tem a desvantagem que você muitas vezes usa *variáveis globais*. Ao invés disso, nós recomendamos definir um procedimento via **proc() begin ... end_proc.** E$ste conceito da linguagem de programação do MuPAD é muito mais flexível e é discutido em mais detalhes no Capítulo 18.

Exercício 4.31: Defina as funções *f (x) = x2* e *g (x)* = $\sqrt{x}$. Compute *f (f (g(2))* e $\underbrace{f(f(\ldots f}_{100\ \text{times}}(x)..)).$

Exercício 4.32: Defina a fu8nção que contraria a ordem dos elementos em uma lista.

Exercício 4.33: Os *polinômios de Chebyshev* são definidos recursivamente pela seguinte fórmula:

$T_0(x) = 1,\ T_1(x) = x,\ T_k(x) = 2x\,T_{k-1}(x) - T_{k-2}(x).$

Compute os valores de $T_2(x)$,..., $T_5(x)$ para $x = 1/3$, $x = 0{,}33$, e para um valor simbólico *x*.

4.13 Expansões de séries

Expressões tais como 1/ (1 – x) admitem expansões de séries para parâmetros simbólicos. Este exemplo particularmente simples é a soma das séries geométricas:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \ldots .$$

A função **taylor** computa os termos conduzidos de tais séries:

```
>> t := taylor(1/(1 - x), x = 0, 9)
```
$$1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + O\left(x^9\right)$$

Esta são as séries de expansão da expressão em volta do ponto $x = 0$, como requisitado pelo segundo argumento. MuPAD tem truncado as séries infinitas antes do termo x^9 e coletou a fileira no "grande Oh" termo $O\ (x^9)$ (também chamado "símbolo Laudau"). O terceiro argumento (opcional) de **Taylor** controla a mutilação. Se isso não é apresentado, então MuPAD substitui o valor da variável ambiente ORDER ao invés disso, a qual o valor padrão é 6:

```
>> t := taylor(1/(1 - x), x = 0)
```
$$1 + x + x^2 + x^3 + x^4 + x^5 + O\left(x^6\right)$$

As séries resultantes parecem como uma soma ordinária com um termo adicional () (.). Internamente, contudo, isso é representado por uma estrutura especial de dados do tipo de domínio **Series::Puiseux**

```
>> domtype(t)
```
Series::Puiseux

O grande termo Oh por si só é uma estrutura de dados por si mesmo, do tipo de domínio () e com regras especiais de manipulação:

```
>> 2*O(x^2) + O(x^3), x^2*O(x^10), O(x^5)*O(x^20),
   diff(O(x^3), x)
```
$$O\left(x^2\right),\ O\left(x^{12}\right),\ O\left(x^{25}\right),\ O\left(x^2\right)$$

A ordem dos termos nas séries de Taylor é fixada: potências com pequenos expoentes procedem estes com altos expoentes. Isto está em contraste com a ordem em somas ordinárias, onde altos expoentes precedem pequenos expoentes:

```
>> S := expr(t)
```
$$x^5 + x^4 + x^3 + x^2 + x + 1$$

Aqui nós usamos a função do sistema **expr** para converter as séries para uma expressão do tipo de domínio DOM_EXPR. Como você pode ver na saída, o O (.) termo foi cortado fora.

O comando **op** atua em séries em um modo não óbvio e não deve ser usado:

```
>> op(t)
```
$$0,\ 1,\ 0,\ 6,\ [1, 1, 1, 1, 1, 1],\ x = 0,\ \text{Undirected}$$

O primeiro operando é um tipo bandeira (ou 0 ou 1) usado para propósitos internos. O segundo operando é o "grau ramificando" e provê informação sobre a não singularidade da expansão. O terceiro e o quarto operando denota o menor expoente na expansão e no *o* termo, respectivamente. O quinto operando é uma lista com os coeficientes. Os sextos operandos consistem de uma equação com a expansão variável e o ponto de expansão. O último operando é uma bandeira interna indicando se as séries são válidas em algum a vizinhança do ponto de expansão ou apenas para a esquerda ou a direita do ponto de expansão ao longo dos eixos.

O usuário não precisa conhecer tais internos. Então, a função **coeff** provê extrair os coeficientes. Isto é mais intuitivo que **op.** A chamada **coeff (t, i)** retorna o coeficiente de x^i:

```
>> t := taylor(cos(x^2), x, 20)
```
$$1 - \frac{x^4}{2} + \frac{x^8}{24} - \frac{x^{12}}{720} + \frac{x^{16}}{40320} - \frac{x^{20}}{3628800} + O\left(x^{24}\right)$$
```
>> coeff(t, 0), coeff(t, 1), coeff(t, 20), coeff(t, 25)
```
$$1,\ 0,\ -\frac{1}{3628800},\ \text{FAIL}$$

No exemplo anterior, nós provimos **x** como segundo argumento para especificar o ponto da expansão. Isso é equivalente a **x = 0.**

As operações aritméticas usuais também trabalham para séries:

```
>> a := taylor(cos(x), x, 3): b := taylor(sin(x), x, 4):
>> a, b
```
$$1 - \frac{x^2}{2} + O\left(x^4\right),\ x - \frac{x^3}{6} + O\left(x^5\right)$$
```
>> a + b, 2*a*b, a^2
```
$$1 + x - \frac{x^2}{2} - \frac{x^3}{6} + O\left(x^4\right),\ 2\,x - \frac{4\,x^3}{3} + O\left(x^5\right),\ 1 - x^2 + O\left(x^4\right)$$

Ambos os operadores de composição @ e o operador repetição @@ aplicam-se para séries também:

```
>> a := taylor(sin(x), x, 20):
>> b := taylor(arcsin(x), x, 20): a@b
```
$$x + O\left(x^{21}\right)$$

Se você tentar computar as séries de Taylor de uma função que não tem uma, então **taylor** aborta com um erro. A função **series** pode computar expansões mais gerais (séries de Laurent, séries de Puiseux):

```
>> taylor(cos(x)/x, x = 0, 10)
   Error: does not have a Taylor series expansion, try \
   'series' [taylor]

>> series(cos(x)/x, x = 0, 10)
```

$$\frac{1}{x} - \frac{x}{2} + \frac{x^3}{24} - \frac{x^5}{720} + \frac{x^7}{40320} + O\left(x^9\right)$$

Você pode gerar séries de expansões nos termos de potências negativas expandindo em volta do ponto **infinity:**

```
>> series((x^2 + 1)/(x + 1), x = infinity)
```

$$x - 1 + \frac{2}{x} - \frac{2}{x^2} + \frac{2}{x^3} - \frac{2}{x^4} + O\left(\frac{1}{x^5}\right)$$

Isto é um exemplo de expansão "assintótica", com aproximados o comportamento da função para valores extensos do argumento. Em casos simples, **series** retorna uma expansão em termos de potências negativas de *x*, mas outras funções podem tornar se também:

```
>> series((exp(x) - exp(-x))/(exp(x) + exp(-x)),
          x = infinity)
```

$$1 - \frac{2}{(e^x)^2} + \frac{2}{(e^x)^4} - \frac{2}{(e^x)^6} + \frac{2}{(e^x)^8} - \frac{2}{(e^x)^{10}} + O\left(\frac{1}{(e^x)^{12}}\right)$$

Exercício 4.34: A ordem *p* de uma raiz *x* de uma função *f* é o número máximo de derivadas que desaparecem no ponto *x*: $f(x) = f'(x) = \ldots = f^{(p-1)}(x) = 0, f^{(p)}(x) \neq 0$.
Qual é a ordem da raiz $x = 0$ de $f(x) = \tan(\sin(x)) - \sin(\tan(x))$?

Exercício 4.35: Além dos operadores aritméticos, algumas outras funções do sistema como **diff** ou **int** trabalham diretamente para séries. Compare o resultado de **taylor (diff (1 – x), x, x)** e a derivada de **taylor (1/ (1 – x), x)**. Matematicamente, ambas as séries são idênticas. Você pode explicar a diferença no MuPAD?

Exercício 4.36: A função $f(x) = \sqrt{x+1} - \sqrt{x-1}$ tende a zero para largo *x*, por exemplo, $\lim_{x \to \infty} f(x) = 0$. Mostre que a aproximação $f(x) \approx 1/\sqrt{x}$ é válida para largos valores de *x*. Encontre aproximações melhor assintóticas de *f*.

Exercício 4.37: Compute os primeiros três termos nas séries de expansões da função **f:= sin (x + x^3)** em volta de **x = 0**. Leia a página de ajuda para a função do MuPAD **revert**. Use esta função para computar os termos conduzidos das séries de expansão da função inversa f^{-1}(a qual é também definida em uma certa vizinhança de $x = 0$).

4.14 Estruturas algébricas: campos, anéis, etc.

O kernel do MuPAD provê tipos de domínio para as estruturas de dados básicas tais como números, conjuntos, tabelas, etc. Em adição, você pode definir sua própria estrutura de dados na linguagem do MuPAD e trabalhar com elas simbolicamente. Nós não discutimos a construção de tais novos "domínios" nesta introdução elementar, mas demonstramos algumas "bibliotecas" especiais domínios providos pelo sistema.

Por trás dos domínios do kernel, a biblioteca **Dom** contém uma variedade de domínios pré-definidos que foram implementados pelos desenvolvedores do MuPAD. O seguinte comando uma avaliação:

```
>> info(Dom)
   Library 'Dom': basic domain constructors

   -- Interface:
   Dom::AlgebraicExtension,
   Dom::ArithmeticalExpression,
   Dom::BaseDomain,
   Dom::Complex,
   ...
```

As páginas de ajuda tais como **?Dom::Complex** proveem informação em estruturas de dados individuais. Nesta seção, nós apresentamos alguns domínios particularmente úteis representando objetos matemáticos complexos tais como campos, anéis, etc. Seção **4.15** discute um tipo de dado para matrizes, a qual é bem encaixada para problemas em álgebra linear.
A principal parte de um domínio é seu *construtor*, o qual gera objetos do domínio. Cada tal objeto "conhece" seu domínio, o qual tem *métodos* preso a eles. Eles representam as operações matemáticas para esses objetos.

Aqui está um a lista de algumas das bem conhecidas estruturas matemáticas implementadas na biblioteca **Dom:**

- o anel das integrais $\mathbb{Z}$: **Dom::Integer**,

- o campo dos números racionais $\mathbb{Q}$: **Dom::Rational**,

- o campo dos números reais $\mathbb{R}$: **Dom::Real** ou **Dom::Float**,

- o campo dos números complexos $\mathbb{C}$: **Dom::Complex**,

- o anel das integrais de módulo *n*: **Dom::IntegerMod (n)**.

Nós consideramos a classe de resíduo de integrais módulo *n*. Estes elementos são as integrais 0, 1,..., *n* – 1, e adição e multiplicação são definidas “modulo *n*.” Isso trabalha adicionando ou multiplicando em $\mathbb{Z}$, dividindo o resultado por *n*, e tomando o resto em {0, 1, ..., *n* – 1 } desta divisão:

```
>> 3*5 mod 7
   1
```

Neste exemplo, nós usamos os tipos de dados do kernel do MuPAD: o operador * multiplica as integrais 3 e 5 no modo ususal para obter 15, e o operador **mod** computa a decomposição 15 = 2 . 7 + 1 e retorna 1 como o resto de módulo 7

Alternativamente, você pode dizer ao MuPAD que você quer computar em $\mathbb{Z}_7 = \mathbb{Z} / 7\,\mathbb{Z}$ usando a sintaxe de entrada **Dom::IntegerMod (7)** Os próximos objetos atuam como um construtor de elementos do resíduo do anel de classe modulo 7:

```
>> constructor := Dom::IntegerMod(7):
>> x := constructor(3); y := constructor(5)
   3 mod 7

   5 mod 7
```

Como você pode ver da tela de saída, os identificadores **x** e **y** não tem as integrais 3 e 15, respectivamente, como valores. Ao invés disso, os números são elementos da classe de anel residual das integrais de módulo 7:

```
>> domtype(x), domtype(y)
   Z7, Z7
```

Agora, você pode usar as operações aritméticas usuais, e MuPAD automaticamente usa as regras de computação do anel de classe residual:

```
>> x*y, x^123*y^17 - x + y
   1 mod 7, 6 mod 7
```

O anel **Dom::IntegerMod (7)** mesmo tem uma estrutura de campo, assim você pode dividir por todos os elementos exceto 0 mod 7:

```
>> x/y
   2 mod 7
```

Um exemplo mais abstrato é a extensão de campo

$$K = \mathbb{Q}[\sqrt{2}] = \{p + q\sqrt{2}; \overline{p}, q \in \mathbb{Q}\}.$$

Você pode definir este campo no MuPAD via

```
>> K := Dom::AlgebraicExtension(Dom::Rational,
                                Sqrt2^2 = 2, Sqrt2):
```

Aqui o identificador **Sqrt2** $(\hat{=}\sqrt{2})$, definido por sua propriedade algébrica Sqrt2^2 = 2, é usado para estender os números racionais **Dom::Rational.** Agora, você pode computar neste campo:

```
>> x := K(1/2 + 2*Sqrt2): y := K(1 + 2/3*Sqrt2):
>> x^2*y + y^4
   677 Sqrt2    5845
   --------- + ----
      54        324
```

O domínio **Dom::ExpressionField (normalizer, zerotest)** representa o “campo” de expressões (simbólicas) do MuPAD. O construtor é parametrizado por duas funções **normalizer** e **zerotest**, as quais podem ser escolhidas pelo usuário.

A função **zerotest** é chamada internamente por todos os algoritmos que quer decidir se um domínio de objeto é matematicamente 0. Tipicamente, você vai usar a função do sistema **iszero**, que reconhece não apenas a integral 0 como zero, mas também outros objetos tais como o número ponto flutuante 0.0 ou o polinômio (0, [x]) (Seção 4.16).

A tarefa da função **normalizer** é gerar uma forma normal para objetos do MuPAD do tipo **Dom::ExpressionField(. , .).** Operações em tais objetos vão usar esta função para simplificar o resultado antes de retornar a ele. Por exemplo, se você fornece a função identidade **id** para o argumento **normalizer**, então operações em objetos deste domínio trabalham como para as expressões ususais do MuPAD sem normalização adicional:

```
>> constructor := Dom::ExpressionField(id, iszero):
>> x := constructor(a/(a + b)^2):
   y := constructor(b/(a + b)^2):
>> x + y
      a          b
   ------- + -------
   (a+b)^2    (a+b)^2
```

Se você fornecer a função do sistema **normal** ao invés, o resultado é simplificado automaticamente (Seção 9.1):

```
>> constructor := Dom::ExpressionField(normal, iszero):
>> x := constructor(a/(a + b)^2):
   y := constructor(b/(a + b)^2):
```

```
>> domtype(Dom::Integer(2)),
   domtype(Dom::Rational(2/3)),
   domtype(Dom::Float(PI)),
   domtype(Dom::ExpressionField(id, iszero)(a + b))
    DOM_INT, DOM_RAT, DOM_FLOAT, DOM_EXPR
```

Nós notamos que o propósito de tais domínios do MuPAD não são necessariamente a geração direta de estruturas de dados ou a computação com os objetos correspondentes. Realmente, alguns construtores simplesmente retornam objetos dos domínios que estão por baixo do kernel, se tais domínios existem:
Nestes casos, não há beneficio imediato em usar tal construtor; você pode também computar diretamente com os correspondentes objetos do kernel. A principal aplicação de tais estruturas de dados especiais é a construção de estruturas matemáticas mais complexas. Um simples exemplo é a geração de matrizes (Seção 4.15) ou polinômios (Seção 4.16) com entradas em um anel particular, tais que a matriz ou aritmética polinomial, respectivamente, é executada de acordo com as regras computacionais do anel de coeficiente.

4.15 Vetores e matrizes

Na Seção 4.14, nós demos exemplos de tipos de dados especiais (“domínios”) para definir estruturas algébricas tais como anéis, campos, etc. no MuPAD. Nessa seção, nós discutimos dois domínios futuros para geração e computação conveniente com vetores e matrizes: **Dom::Matrix** e **Dom::SquareMatrix**. Em princípio, você pode usar arranjos para trabalhar com vetores e matrizes (seção 4.9). Entretanto, então você tem que definir suas próprias rotinas para adição, multiplicação, inversão, ou computação de determinantes, usando a linguagem de programação do MuPAD (Capítulo 18). Para um tipo especial de matriz que nós apresentamos no que se segue, tais rotinas existem e são “presas” às matrizes como métodos. Mais ainda, você pode usar a função da biblioteca **linalg** (álgebra linear, Seção 4.15.4), que pode controlar matrizes deste rtipo.

4.15.1 Definição de matrizes e vetores

Vetores no MuPAD são considerados como matrizes especiais de dimensão 1 x *n* ou
n x 1, respectivamente. O comando **matrix** pode ser usado para criar matrizes e vetores de dimensão arbitrária:

```
>> matrix([[  1,      2,      3,     4   ],
           [  a,      b,      c,     d   ],
           [sin(x), cos(x), exp(x), ln(x)]]),
   matrix([x1, x2, x3, x4])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ a & b & c & d \\ \sin(x) & \cos(x) & e^x & \ln(x) \end{pmatrix}, \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \end{pmatrix}$$

Expressões aritméticas arbitrárias podem ser usadas como elementos. Embora matrizes criadas por **matrix** são satisfatórias para a maioria das aplicações e vai ser mais amplamente usados objetos de matriz. De fato, a função **matrix** é só um construrtor com um nome curto para matrizes especiais do tipo de domínio **Dom::Matrix (R)**, com um anel de coeficiente especial R que representa arbitrariamente expressões no MuPAD.

Nós explicamos o conceito geral. MuPAD provê o tipo de dado **Dom::Matrix** para matrfizes de dimensão arbitrária *m* x *n*. O tipo **Dom::SquareMatrix** representa matrizes quadradas de dimensão *n* x *n*. Elas pertencem a biblioteca **Dom**, que também cumpre tipos de dados para estruturas matemáticas tais como anéis e campos. (Seção 4.14). Matrizes podem ter entradas de um conjunto que devem ser equipadas com uma estrutura de anel no senso matemático. Por exemplo, você pode usar os anéis pré-definidos e campos tais como **Dom::Integer**, **Dom::IntegerMod (n)**, etc. da biblioteca **Dom**.
A chamada **Dom::Matrix (R)** cria o construtor para matrizes de dimensão arbitrária
m x *n* com coeficientes no anel R. Quando você constrói uma tal matriz, você está requerido assegurar que estas entradas pertençam à (ou pode ser convertido para) este anel. Você deve guardar isto em mente quando tenta gerar matrizes com entradas fora do anel de coeficiente em uma computação (por exemplo, o inverso de uma integral matriz em geral tem entradas não integrais racionais).

O seguinte exemplo rende o construtor para matrizes com entradas de números racionais:

```
>> constructor := Dom::Matrix(Dom::Rational)
```

$$\mathrm{Dom::Matrix}(\mathbb{Q})$$

Agora, você pode gerar matrizes de dimensões arbitrárias. No seguinte exemplo, nós geramos uma matriz 2 *X* 3 com todas as entradas inicializadas em 0:

```
>> A := constructor(2, 3)
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Quando gera uma matriz, você pode fornecer uma função *f* que leva dois argumentos. Então a entrada é agora na fileira *i* e coluna *j* é inicializada com *f (i, j):*

```
>> f := (i, j) -> (i*j):  A := constructor(2, 3, f)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \end{pmatrix}$$

Alternativamente, você pode iniciar uma matriz por especificar lista (aninhada). Cada elemento da lista é por si só uma lista e corresponde a uma fileira da matriz. O comando seguinte gera a mesma matriz como no exemplo anterior:

```
>> constructor(2, 3, [[1, 2, 3], [2, 4, 6]]):
```

Os parâmetros para a dimensão são opcionais aqui, desde que eles também são dados pela estrutura da lista. Assim,

```
>> constructor([[1, 2, 3], [2, 4, 6]]):
```

Também retornam a mesma matriz. Um arranjo do tipo DOM_ARRAY (Seção 4.9) também é válido para iniciar uma matriz:

```
>> Array := array(1..2, 1..3, [[1, 2, 3], [2, 4, 6]]):
>> Matrix := constructor(Array):
```

Você pode definir vetores colunas e fileiras como matrizes *m x* 1 e 1 x *n*, respectivamente. Listas planas podem ser usadas para definir vetores coluna ou fileira:

```
>> column := constructor(3, 1, [1, 2, 3])
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

```
>> row := constructor(1, 3, [1, 2, 3])
```

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

Se a dimensão não está especificada, a lista plana gera um vetor coluna:

```
>> column := constructor([1, 2, 3])
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

As entradas de uma matriz ou um vetor podem ser acessadas nas formas **matrix [i, j]**, **row [i]**, ou **column [j]**.Desde que vetores são matrizes especiais, você pode acessar os componentes de um vetor também na forma **row [1, i]** ou **column [j, i]**, respectivamente:

```
>> A[2, 3], row[3], row[1, 3],
   column[2], column[2, 1]
```

$$6,\ 3,\ 3,\ 2,\ 2$$

Submatrizes são geradas como segue:

```
>> Matrix[1..2, 1..2], row[1..1, 1..2],
   column[1..2, 1..1]
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}, \begin{pmatrix} 1 & 2 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

Você pode mudar uma entrada de matriz por uma alegação indexada:

```
>> Matrix[2, 3] := 23: row[2] := 5: column[2, 1] := 5:
>> Matrix, row, column
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 23 \end{pmatrix}, \begin{pmatrix} 1 & 5 & 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 5 \\ 3 \end{pmatrix}$$

Você pode usar loops (Capítulo 16) para mudar todos os componentes de um a matriz:

```
>> m := 2: n := 3: Matrix := constructor(m, n):
>> for i from 1 to m do
     for j from 1 to n do
       Matrix[i, j] := i*j
     end_for
   end_for:
```

Você pode gerar matrizes diagonais fornecendo a opção **Diagonal.** Neste caso, o terceiro argumento para o construtor pode ou ser uma lista dos elementos diagonais de uma função *f* tal que o *i*º elemento diagonal é *f* (*i*):

```
>> constructor(2, 2, [11, 12], Diagonal)
```

$$\begin{pmatrix} 11 & 0 \\ 0 & 12 \end{pmatrix}$$

No próximo exemplo, nós geramos uma matriz identidade fornecendo 1 como uma função definindo os elementos diagonais:

```
>> constructor(2, 2, 1, Diagonal)
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

O construtor considerado tão distante retorna matrizes com racionais (por exemplo, reais) entradas de número. Assim, a seguinte tentativa para gerar uma matriz com coeficientes complexos não trabalha:

```
>> constructor([[1, 2, 3], [2, 4, 1 + I]])
   Error: unable to define matrix over Dom::Rational \
```

```
>> constructor := Dom::Matrix(Dom::Complex):
>> constructor([[1, 2, 3], [2, 4, 1 + I]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 1+\mathrm{i} \end{pmatrix}$$

Você tem que escolher um anel coeficiente satisfatório para gerar uma matriz com as entradas acima. No próximo exemplo, nós definimos um novo construtor para matrizes com entradas de número complexo:

Você pode gerar matrizes as quais as entradas são expressões arbitrarias do MuPAD pelos sentidos do campo **Dom::ExpressionField (id, iszero)** (veja Seção 4.14). Isto é o anel coeficiente padrão para matrizes. Você pode sempre usar este anel quando os coeficientes e suas propriedades são irrelevantes. **Dom::Matrix ()** sem nenhum argumento é um construtor para tais matrizes. Como um a notação tarquigrafia, este anel matriz é preso ao identificador **matrix:**

```
>> matrix
```

Dom::Matrix()

```
>> matrix([[1, x + y, 1/x^2], [sin(x), 0, cos(x)],
           [x*PI, 1 + I, -x*PI]])
```

$$\begin{pmatrix} 1 & x+y & \frac{1}{x^2} \\ \sin(x) & 0 & \cos(x) \\ \pi\, x & 1+\mathrm{i} & -\pi\, x \end{pmatrix}$$

Se você usar **Dom::ExpressionField (normal, iszero)** como o anel coeficiente, então todas as entradas de matriz são simplificadas via função **normal**, como descrito na Seção 4.14. Operações aritméticas com tais matrizes são comparativamente lentas, desde que uma chamada para **normal** pode levar um pouco mais de tempo. Contudo, os resultados são em geral mais simples que os resultados (equivalentes) das computações com o anel coeficiente padrão **Dom::ExpressionField (id, zero)** usados por **Dom::Matrix ()**.

O construtor **Dom::SquareMatrix (n, R)** corresponde ao anel de matrizes quadradas *n* -dimensionais com o anel coeficiente R. Se o argumento R está faltando, então MuPAD automaticamente usa o anel coeficiente para todas as expressões do MuPAD:

```
>> constructor := Dom::SquareMatrix(2)
```

Dom::SquareMatrix(2)

```
>> constructor([[0, y], [x^2, 1]])
```

$$\begin{pmatrix} 0 & y \\ x^2 & 1 \end{pmatrix}$$

4.15.2 Computando com matrizes

Você pode usar os operadores aritméticos padrões para fazer aritmética básica com matrizes:

```
>> A := matrix([[1, 2], [3, 4]]):
>> B := matrix([[a, b], [c, d]]):
>> A + B, A*B;
   A*B - B*A, A^2 + B
```

$$\begin{pmatrix} a+1 & b+2 \\ c+3 & d+4 \end{pmatrix}, \begin{pmatrix} a+2c & b+2d \\ 3a+4c & 3b+4d \end{pmatrix}$$

$$\begin{pmatrix} 2c-3b & 2d-3b-2a \\ 3a+3c-3d & 3b-2c \end{pmatrix}, \begin{pmatrix} a+7 & b+10 \\ c+15 & d+22 \end{pmatrix}$$

Multiplicação de uma matriz e um número trabalha modo componente (multiplicação escalar):

```
>> 2*B
```

$$\begin{pmatrix} 2a & 2b \\ 2c & 2d \end{pmatrix}$$

O inverso de uma matriz é representado por **1/A ou A ^(-1)**:

```
>> C := 1/A
```

$$\begin{pmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{pmatrix}$$

Um simples teste mostra que o inverso computado está correto:

```
>> A*C, C*A
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Uma inversão retorna **FAIL** quando MuPAD é incapaz de computar o resultado. A seguinte matriz não é reversiva:

```
>> C := matrix([[1, 1], [1, 1]]): C^(-1)
   FAIL
```

O operador concatenação ., que combina listas (Seção 4.6) ou fios (Seção 4.11), é “sobrecarregado” para matrizes. Você pode usar ele para combinar matrizes com o mesmo número de fileiras:

```
>> A, B, A.B
```

$$\left(\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array}\right), \left(\begin{array}{cc} a & b \\ c & d \end{array}\right), \left(\begin{array}{cccc} 1 & 2 & a & b \\ 3 & 4 & c & d \end{array}\right)$$

Por trás dos operadores aritméticos, outros sistemas de funções são aplicáveis para matrizes. Aqui está uma lista de exemplos

- **conjugate (A)** substitui todos os componentes por seus conjugados complexos,
- **diff (A, x)** difreencia componentes com respeito a **x**,
- **exp (A)** computa $e^A = \sum_{i=0}^{\infty} \frac{1}{i!} A^i$,
- **expand (A)** aplica **expand** para todos os componentes de **A**,
- **expr (A)** converte **A** para um arranjo do tipo de domínio **DOM_ARRAY**,
- **float (A)** aplica **float** para todos os componentes de **A**,
- **has (A, expression)** checa se uma expressão é contida em ao menos uma entrada de **A**,
- **int (A, x)** integra conhecimento de componente com respeito a **x**,
- **iszero (A)** checa se todos os componentes de **A** somem,

- **map (A, function)** aplica a função para todos os componentes,

- **norm (A)** (idêntico com **norm (A, Infinity))** computa a norma infinita,

- **subs (A, equation)** aplica **subs (. , equation)** para todas as entradas de **A**,
- C:=**zip (A, B, f)** retorna a matriz definida por $C_{ij} = f(A_{ij}, B_{ij})$.

A biblioteca álgebra linear **linalg** e as bibliotecas numéricas **numeric** (Seção 4.15.4) cumpre tantas outras funções de controle de matrizes.

Exercício 4.38: Gere a matriz 15 × 15 de Hilbert H (H_{ij}) com $H_ij = 1/(i + j - 1)$. Gere o vetor $\overline{b} = He\rightarrow$, onde $e\rightarrow = (1, \ldots, 1)$. Gere a solução exata vetorial $x\rightarrow$ do sistema de equações $H\, x\rightarrow = \overline{b}$ (de certo, isso deve pertencer $x\rightarrow = e\rightarrow$). Converta todas as entradas de H para valores de pontos flutuantes e resolva o sistema de equações novamente. Compare os resultados para a solução exata. Você notará uma diferença dramática, a qual é causada por erros de arredondamento numérico. Matrizes largas de Hilbert não podem ser invertidas com a precisão padrão de software numérico comum!

4.15.3 Métodos especiais para matrizes

Um construtor que já foi gerado pelos sentidos de ou **Dom::Matrix** (.) ou **Dom::SquareMatrix** (.) contém muitas funções especiais para o tipo de dado correspondente. Se M:= Dom::Matrix (ring) é um construtor e A:=M (.) é um a matriz gerada com este construtor, como descrito na Seção 4.15.1, então os seguintes métodos são disponíveis:
(tabela a fazer depois)

```
>> M := Dom::Matrix(): A := M([[x, 1], [2, y]])
```

$$\left(\begin{array}{cc} x & 1 \\ 2 & y \end{array}\right)$$

```
>> M::col(A, 1), M::delCol(A, 1), M::matdim(A)
```

$$\left(\begin{array}{c} x \\ 2 \end{array}\right), \left(\begin{array}{c} 1 \\ y \end{array}\right), [2, 2]$$

```
>> M::swapCol(A, 1, 2), M::tr(A), M::transpose(A)
```

$$\left(\begin{array}{cc} 1 & x \\ y & 2 \end{array}\right), x + y, \left(\begin{array}{cc} x & 2 \\ 1 & y \end{array}\right)$$

Desde que o domínio é preso ao objeto A como A::dom, um pode também ser chamado os métodos de domínio via A::dom::method. Por exemplo:

```
>> A::dom::tr(A)
```

$$x + y$$

A função **info** retorna um guia de sobrevivência destes métodos:

```
>> info(Dom::Matrix())
   -- Domain:
   Dom::Matrix()

   -- Constructor:
   Dom::Matrix

   -- Super-Domains:
   Dom::BaseDomain

   -- Categories:
   Cat::Matrix(Dom::ExpressionField()), Cat::BaseCategor\
   y

   -- No Axioms.

   -- Entries:
   Content, Name, Simplify, TeX, _concat, _divide, _inde\
   x, _invert, _mod, _mult, _mult1, _mult2, _multNC1, _m\
   ultNC2, _negate, _plus, _power, _subtract, addCol, ad\
   dRow, allAxioms, allCategories, allEntries, allSuperD\
   omains, assignElements, coeffRing, coerce, col, conca\

   tMatrix, conjugate, convert, convert_to, cos, create,\
    create_dom, delCol, delRow, diff, doprint, equal, eq\
   uiv, exp, expand, expr, expr2text, factor, float, fou\
   rier, gaussElim, getAxioms, getCategories, getSuperDo\
   main, has, hasProp, identity, indets, info, int, invf\
   ourier, invlaplace, is, isDense, isSparse, iszero, ke\
   y, kroneckerProduct, laplace, length, map, mapNonZero\
   es, mapcoeffs, matdim, mkSparse, modp, mods, multCol,\
    multRow, multcoeffs, new, nonZeroOperands, nonZeroes\
   , nonZeros, nops, norm, normal, op, print, printMaxSi\
   ze, printMethods, random, randomDimen, row, setCol, s\
   etPrintMaxSize, setRow, set_index, simplify, sin, sol\
   ve, stackMatrix, subs, subsex, subsop, swapCol, swapR\
   ow, testeq, testtype, tr, transpose, unapply, undefin\
   edEntries, whichEntry, zip
```

Abaixo das legendas **Entries**, você encontra uma lista de todos os métodos do domínio. A chamada ?**Dom::Matrix** retorna uma descrição completa destes métodos.

4.15.4 As bibliotecas linalg e numeric

Por trás dois sistemas de funções operando em matrizes, a biblioteca **linalg** contém uma variedade de outras funções de álgebra linear:

-- Interface:
linalg::addCol, linalg::addRow, linalg::adjoint,
linalg::angle, linalg::basis, linalg::charmat,
linalg::charpoly, linalg::col, linalg::companion,
linalg::concatMatrix, linalg::crossProduct, linalg::curl,
linalg::delCol, linalg::delRow, linalg::det,
linalg::divergence, linalg::eigenvalues, linalg::eigenvectors,
linalg::expr2Matrix, linalg::factorCholesky, linalg::factorLU,
linalg::factorQR, linalg::frobeniusForm, linalg::gaussElim,
linalg::gaussJordan, linalg::grad, linalg::hermiteForm,
linalg::hessenberg, linalg::hessian, linalg::hilbert,
linalg::intBasis, linalg::inverseLU, linalg::invhilbert,
linalg::invpascal, linalg::invvandermonde, linalg::isHermitean,
linalg::isPosDef, linalg::isUnitary, linalg::jacobian,
linalg::jordanForm, linalg::kroneckerProduct, linalg::laplacian,
linalg::matdim, linalg::matlinsolve,
linalg::matlinsolveLU, linalg::minpoly,
linalg::multCol, linalg::multRow,
linalg::ncols, linalg::nonZeros, linalg::normalize,
linalg::nrows, linalg::nullspace, linalg::ogCoordTab,
linalg::orthog, linalg::pascal, linalg::permanent,
linalg::potential, linalg::pseudoInverse, linalg::randomMatrix,
linalg::rank, linalg::row, linalg::scalarProduct,
linalg::setCol, linalg::setRow, linalg::smithForm,
linalg::stackMatrix, linalg::submatrix, linalg::substitute,
linalg::sumBasis, linalg::swapCol, linalg::swapRow,
linalg::sylvester, linalg::toeplitz, linalg::toeplitzSolve,
linalg::tr, linalg::transpose, linalg::vandermonde,
linalg::vandermondeSolve, linalg::vecdim, linalg::vectorOf,
linalg::vectorPotential, linalg::wiedemann

Algumas dessas funções, tais como linalg::col ou linalg::delCol, simplesmente chamam os métodos internos para matrizes que nós descrevemos na Seção 4.15.3, e desde que não adicione nova funcionalidade. Contudo, **linalg** também contém muitos algoritmos adicionais. O comando ?linalg rende uma curta descrição de todas as funções. Você pode encontrar uma descrição detalhada de uma função na página de ajuda correspondente, por exemplo:

```
>> ?linalg::det
```

linalg::det -- *determinant of a matrix*

Introduction

linalg::det(A) computes the determinant of the square matrix A.

Call(s)

linalg::det(A)

Parameters

A - a square matrix of a domain of category *Cat::Matrix*.

...

Você pode usar o nome caminho completo library::function para chamar uma função:

```
>> A := matrix([[a, b], [c, d]]): linalg::det(A)
```

$$a\,d - b\,c$$

O polinômio característico det (*x* E – A) de uma matriz é

```
>> linalg::charpoly(A, x)
```

$$x^2 + (-a - d)\,x + a\,d - b\,c$$

Os autovalores são

```
>> linalg::eigenvalues(A)
```

$$\left\{\frac{a}{2}+\frac{d}{2}-\frac{\sqrt{a^2-2\,a\,d+d^2+4\,b\,c}}{2}, \frac{a}{2}+\frac{d}{2}+\frac{\sqrt{a^2-2\,a\,d+d^2+4\,b\,c}}{2}\right\}$$

A biblioteca numérica **numeric** (veja **?numeric**) contém várias funções para computações numéricas com matrizes:

`numeric::det`	: determinant
`numeric::expMatrix`	: exp(Matrix)
`numeric::factorCholesky`	: Cholesky factorization
`numeric::factorLU`	: LU factorization
`numeric::factorQR`	: QR factorization
`numeric::fMatrix`	: functional calculus
`numeric::inverse`	: inversion
`numeric::eigenvalues`	: eigenvalues
`numeric::eigenvectors`	: eigenvalues and –vectors
`numeric::singularvalues`	: singular values
`numeric::singularvectors`	: singular values and vectors

Parcialmente, estas rotinas trabalham para matrizes com entradas simbólicas do tipo **Dom::ExpressionField** e então são mais eficientes para matrizes compridas como as funções **linalg**. Entretanto, o depois pode controlar arbitrariamente anéis coeficientes.

Exercício 4.39: Encontre os valores de *a*, *b*, *c* para tal matriz $\begin{pmatrix} 1 & a & b \\ 1 & 1 & c \\ 1 & 1 & 1 \end{pmatrix}$

Exercício 4.40: Considere as seguintes matrizes:

$$A = \begin{pmatrix} 1 & 3 & 0 \\ -1 & 2 & 7 \\ 0 & 8 & 1 \end{pmatrix}, B = \begin{pmatrix} 7 & -1 \\ 2 & 3 \\ 0 & 1 \end{pmatrix}$$

Deixe B^T ser o transposto de B. Compute o inverso de 2 A+B B^T, ambos acima dos números racionais e acima da classe residual de módulo 7.

Exercício 4.41: Gere a matriz 3 x 3

$$A_{ij} = \{0 \text{ para } i = j,\ 1 \text{ para } i \neq j.$$

Compute este determinante, seu polinômio característico, e seus autovalores. Para cada autovalor, compute uma base do autoespaço correspondente.

4.15.5 Matrizes escassas

Com a versão 2.5 do MuPAD, um tipo especial de matriz Dom::SparseMatrix para representar matrizes escassas foi introduzido. Isso serviu para computar eficientemente com (largas) matrizes que tem apenas uns poucos elementos não zeros. Com a versão do MuPAD 3.0, a representação interna para as matrizes usuais criadas por **matrix** ou **Dom::Matrix(R)** foi adaptada. Agora, estas “matrizes padrão” servem para representar ambas densas também como matrizes escassas e o usuário não tem que se preocupar em usar diferentes tipos de matrizes. O domínio Dom::SparseMatrix do MuPAD 2.5 se torna obsoleto.

Aqui estão algumas observações relativas a eficiência quando suas matrizes forem grandes e escassas:

- Use o anel coeficiente padrão **Dom::ExpressionField**() de exzpressões arbitrárias do MuPAD sempre que possível. Como discutido na seção anterior, o construtor **Dom::Matrix** () cria matrizes deste tipo. Por conveniência, este construtor também está disponível como a função **matrix.**

- Leituras e escritas indexadas para matrizes grandes é um pouco caro. Se possível, um deve esvaziar criando grandes matrizes vazias da dimensão desejada por **matrix (m, n)** e preenchendo nos entradas não zeros pelas tarefas indexadas. Um deve jogar as entradas diretamente quando cria a matriz.

Por exemplo, listas de equações podem ser usadas para especificar as entradas quando cria uma matriz. A seguinte matriz *A* de dimensão 1000 x 1000 consiste de uma faixa diagonal e duas entradas adicionais no topo à direita e à esquerda abaixo. Nós mostramos algumas entradas desta 10ª potência:

```
>> n := 1000:
   A := matrix(n, n, [(i, i) = i $ i = 1..n,
                      (n, 1) = 1, (1, n) = 1]):
   B := A^10:
   B[1, 1], B[1, n]
    1002010022050086122130089, 1001009015040066101119105055
```

No seguinte exemplo, nós geramos uma matriz tri diagonal Toeplitz *A* 100 x 100 e resolvemos a equação A_x = b, onde b é o vetor coluna (1,...,1):

```
>> A := matrix(100, 100, [-1, 2, -1], Banded):
   b := matrix(100, 1, [1 $ 100]):
   x := (1/A)*b
```

$$\begin{pmatrix} 50 \\ 99 \\ \dots \\ 50 \end{pmatrix}$$

Aqui nós aplicamos o inverso **1/A** de **A** ao lado direito da equação. Note, contudo, que invertendo uma matriz escassa não é uma boa ideia porque o inverso de uma matriz escassa é, em geral, não escasso. É muito mais eficiente usar fatoração com matriz escassa para computar a solução de um sistema escasso de equações lineares. Nós usamos o resolvido **numeric::matlinsolve** com a opção Symbolic para computar a solução de um sistema escasso representando 1000 equações para 1000 incógnitas. A rotina **numeric** emprega escassez em um modo opcional:

```
>> A := matrix(1000, 1000, [-1, 2, -1], Banded):
   b := matrix(1000, 1, [1 $ 1000]):
   [x, kernel] := numeric::matlinsolve(A, b, Symbolic):
```

Nós mostramos apenas uns poucos componentes do vetor solução:

```
>> x[1], x[2], x[3], x[4], x[5], x[999], x[1000]
    500, 999, 1497, 1994, 2490, 999, 500
```

4.15.6 Uma aplicação

Nós queremos computar a solução simbólica *a(t), b(t)* do sistema de segunda ordem de equações diferenciais

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\, a(t) = 2\,c\,\frac{\mathrm{d}}{\mathrm{d}t}\, b(t)\,, \quad \frac{\mathrm{d}^2}{\mathrm{d}t^2}\, b(t) = -2\,c\,\frac{\mathrm{d}}{\mathrm{d}t}\, a(t) + 3\,c^2\, b(t)$$

com uma constante arbitrária *c*.Escrevendo *a' (t)* = $\frac{d}{dt}$ *a(t), b'(t)* = $\frac{d}{dt}$ *b(t),* estas equações podem ser equivalentes escritas como um sistema de primeira ordem de equações diferenciais nas variáveis *x(t) = (a(t), a'(t), b(t),b'(t)):*

$$\frac{\mathrm{d}}{\mathrm{d}t}\, x(t) = A\, x(t), \quad A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2\,c \\ 0 & 0 & 0 & 1 \\ 0 & -2\,c & 3\,c^2 & 0 \end{pmatrix}.$$

A solução deste sistema é dada aplicando a matriz exponencial e^{tA} para a condição inicial *x(0):*

x(t) = e^{tA} x(0).

```
>> delete c, t:
   A := matrix([[0,  1,    0,    0 ],
                [0,  0,    0,   2*c],
                [0,  0,    0,    1 ],
                [0, -2*c, 3*c^2, 0 ]]):
```

Nós usamos a função **exp** para computar B = e^{tA}:

```
>> B := exp(t*A)
```

$$\begin{pmatrix} 1 & \frac{2ie^{-ict}}{c} - 3t - \frac{2ie^{ict}}{c} & 3ie^{ict} - 3ie^{-ict} + 6ct & \frac{2}{c} - \frac{e^{ict}}{c} - \frac{e^{-ict}}{c} \\ 0 & ie^{ict} - ie^{-ict} & \frac{3i}{2}ce^{-ict} - \frac{3i}{2}ce^{ict} & \frac{e^{-ict}}{2} + \frac{e^{ict}}{2} \end{pmatrix}$$

```
>> B := map(B, rewrite, sincos):
```

Nós desejamos reescrever os termos (complexos) exponenciais pelas chamadas simbólicas às funções trigonométricas **sin** e **cos**. Isto pode ser alcançado pela rotina **rewrite** (Seção 9.1). Nós usamos o "alvo" sincos para fazer **rewrite** expresso **exp** por **sin** e **cos**. A função **map** é usada para aplicar o comando para todas as ent6radas em B:

Agora, a matriz B parece até mesmo mais complicada do que antes, por exemplo:

```
>> B[1, 2]
```

$$\frac{2i\,(\cos(ct) - i\sin(ct))}{c} - 3t - \frac{2i\,(\cos(ct) + i\sin(ct))}{c}$$

Para simplificar essas expressões, nós aplicamos a função **expand:**

```
>> B := expand(B)
```

$$\begin{pmatrix} 1 & \frac{4\sin(ct)}{c} - 3t & 6ct - 6\sin(ct) & \frac{2}{c} - \frac{2\cos(ct)}{c} \\ 0 & 4\cos(ct) - 3 & 6c - 6c\cos(ct) & 2\sin(ct) \\ 0 & \frac{2\cos(ct)}{c} - \frac{2}{c} & 4 - 3\cos(ct) & \frac{\sin(ct)}{c} \\ 0 & -2\sin(ct) & 3c\sin(ct) & \cos(ct) \end{pmatrix}$$

Nós declaramos uma condição inicial simbólica para o vetor **x(0)**. A chamada seguinte cria uma matriz 4 x 1 para ser interpretada como um vetor coluna:

```
>> x(0) := matrix([a(0), a'(0), b(0), b'(0)])
```

$$\begin{pmatrix} a(0) \\ a'(0) \\ b(0) \\ b'(0) \end{pmatrix}$$

Assim, a solução simbólica desejada do sistema de equações diferenciais é

```
>> x(t) := B*x(0):
```

A solução das funções *a(t) e b(t)* com as condições iniciais simbólicas *a(0), a'(0) (*=D (a) (0)), *b(0), b'(0) (*=D(b)(0)) são:

```
>> a(t) := expand(x(t)[1])
                                2 D(b)(0)
   a(0) - 6 b(0) sin(c t) + --------- - 3 t D(a)(0) -
                                  c

      2 D(b)(0) cos(c t)   4 D(a)(0) sin(c t)
      ------------------ + ------------------ +
               c                    c

      6 c t b(0)
```

```
>> b(t) := expand(x(t)[3])
```

$$4b(0) - 3b(0)\cos(ct) - \frac{2a'(0)}{c} + \frac{2a'(0)\cos(ct)}{c} + \frac{b'(0)\sin(ct)}{c}$$

Finalmente, nós checamos que as expressões acima realmente resolvem o sistema de equações diferenciais:

```
>> expand(diff(a(t),t,t) - 2*c*diff(b(t),t)),
   expand(diff(b(t),t,t) + 2*c*diff(a(t),t) - 3*c^2*b(t))
    0, 0
```

4.16 Polinômios

Computação com polinômios é uma tarefa importante para um sistema algébrico computacional. Com certeza, você pode realizar um polinômio no MuPAD como uma expressão no senso da Seção 4.4 e usar a aritmética padrão:

```
>> polynomialExpression := 1 + x + x^2:
>> expand(polynomialExpression^2)
```

$$x^4 + 2x^3 + 3x^2 + 2x + 1$$

Entretanto. Ali existe um tipo de dado especial **DOM_POLY** junto com algum kernel e funções da biblioteca, com simplificações tais computações e faze-os mais eficientes.

4.16.1 Definição de polinômios

O sistema de função **poly** gera polinômios

```
>> poly(1 + 2*x + 3*x^2)
```

$$\text{poly}\left(3x^2 + 2x + 1, [x]\right)$$

Aqui nós suprimimos a expressão 1 + 2*x* + 3*x*2 (do tipo de domínio **DOM_EXPR**) para **poly**, a qual converte esta expressão para um objeto novo do tipo de domínio **DOM_POLY**. O indeterminado **[x]** é uma parte fixada desse tipo de dado. Isso é relevante para distinguir entre coeficientes indeterminados e (simbólicos) coeficientes e parâmetros. Por exemplo, se você quer considerar a expressão $a_0 + a_1\,x + a_2x^2$ como um polinômio em *x* com coeficientes a_0, a_1, a_2, então a forma acima da chamada para **poly** não rende o resultado desejado:

```
>> poly(a0 + a1*x + a2*x^2)
```
$$\mathrm{poly}\left(\mathrm{a0} + \mathrm{a1}\,x + \mathrm{a2}\,x^2, [\mathrm{a0}, \mathrm{a1}, \mathrm{a2}, x]\right)$$

Isso não representa um polinômio em x mas um polinômio "multivariado" em quatro indeterminadas x, a_0, a_1, a_2. Você pode especificar as indeterminadas de um polinômio na forma de uma lista como argumento para **poly**. O sistema então considera todos os outros identificadores simbólicos como coeficientes simbólicos:

```
>> poly(a0 + a1*x + a2*x^2, [x])
```
$$\mathrm{poly}\left(\mathrm{a2}\,x^2 + \mathrm{a1}\,x + \mathrm{a0}, [x]\right)$$

Se você não especificar uma lista de indeterminados, então **poly** chama a função **indets** para determinar todos os identificadores simbólicos na expressão e interpreta-os como indeterminados de um polinômio:

```
>> indets(a0 + a1*x + a2*x^2, PolyExpr)
```
$$\{\mathrm{a0}, \mathrm{a1}, \mathrm{a2}, x\}$$

A distinção entre indeterminadas e coeficientes é relevante para a representação do polinômio:

```
>> expression := 1 + x + x^2 + a*x + PI*x^2 - b
```
$$x - b + \pi\,x^2 + a\,x + x^2 + 1$$
```
>> poly(expression, [a, x])
```
$$\mathrm{poly}\left(a\,x + (\pi + 1)\,x^2 + x + (1 - b), [a, x]\right)$$
```
>> poly(expression, [x])
```
$$\mathrm{poly}\left((\pi + 1)\,x^2 + (a + 1)\,x + (1 - b), [x]\right)$$

Você pode ver que MuPAD coleta os coeficientes de iguais potências da indeterminada. Os termos são ordenados de acordo com os expoentes que caem

Ao invés de usar uma expressão, você pode gerar um polinômio especificando uma lista de coeficientes não triviais junto com os respectivos expoentes. O comando **poly (list, [x])** gera o polinômio $\sum_{i\geq 0} a_i\,x^{ni}$, vindo da lista $[[a_0, n_0], [a_1, n_1] \ldots, [a_k, n_k]]$:

```
>> list := [[1, 0], [a, 3], [b, 5]]: poly(list, [x])
```
$$\mathrm{poly}\left(b\,x^5 + a\,x^3 + 1, [x]\right)$$

Se você quer construir um polinômio multivariado desse modo, especifique listas de expoentes para todas as variáveis:

```
>> poly([[3, [2, 1]], [2, [3, 4]]], [x, y])
```
$$\mathrm{poly}\left(2\,x^3\,y^4 + 3\,x^2\,y, [x, y]\right)$$

Reciprocamente, a função **poly2list** converte um polinômio para uma lista de coeficientes e expoentes:

```
>> poly2list(poly(b*x^5 + a*x^3 + 1, [x]) )
```
$$[[b, 5], [a, 3], [1, 0]]$$

Para computações mais abstratas, você pode querer restringir os coeficientes de um polinômio para um certo conjunto (matematicamente: um anel) o qual é representado por uma estrutura de dado especial no MuPAD. Nós já vimos exemplos típicos de anéis na Seção 4.14: as integrais **Dom::Integer**, os números racionais **Dom::Rational**, ou o anel da classe residual **Dom::IntegerMod (n)** de integrais de módulo n. Você pode especificar o anel coeficiente como argumento para **poly**:

```
>> poly(x + 1, [x], Dom::Integer)
```
$$\mathrm{poly}\left(x + 1, [x], \mathbb{Z}\right)$$
```
>> poly(2*x - 1/2, [x], Dom::Rational)
```
$$\mathrm{poly}\left(2\,x - \left(\frac{1}{2}\right), [x], \mathbb{Q}\right)$$
```
>> poly(4*x + 11, [x], Dom::IntegerMod(3))
```
$$\mathrm{poly}\left(x + 2, [x], \mathbb{Z}_3\right)$$

Note que no último exemplo, o sistema automaticamente simplificou os coeficientes de acordo com as regras de computação com integrais de módulo 3:

```
>> 4 mod 3, 11 mod 3
```
$$1,\ 2$$

No seguinte exemplo, **poly** converte os coeficientes para números de pontos flutuantes, como especificado pelo terceiro argumento:

```
>> poly(PI*x - 1/2, [x], Dom::Float)
```
$$\mathrm{poly}\left(3.141592654\,x - 0.5, [x], \mathrm{Dom::Float}\right)$$

Se nenhum anel coeficiente é especificado, então MuPAD por padrão usa o anel **Expr** o qual simboliza arbitrariamente expressões do MuPAD. Neste caso, voc~e pode usar identificadores simbólicos como coeficientes:

```
>> polynomial := poly(a + x + b*y, [x, y]);
   op(polynomial)
```
$$\mathrm{poly}\left(x + b\,y + a, [x, y]\right)$$

$$a + x + b\,y,\ [x, y],\ \mathrm{Expr}$$

Nós resumimos que um polinômio do MuPAD cumpre três partes:

1. uma expressão polinomial da forma $\sum a_{i1i2} \ldots \underset{1}{x}^{i1} \underset{2}{x}^{i2} \ldots$,

2. uma lista de indeterminadas $[x_1, x_2, \ldots]$,

3. o anel coeficiente.

Há os três operandos de um polinômio no MuPAD **p**, o qual pode ser acessado via **op (p, 1)**, **op (p, 2)**, e **op (p, 3)**, respectivamente. Assim, você pode converter vários tipos de domínio tais como polinômios ou expressões:

```
>> polynomial := poly(x^3 + 5*x + 3)
```
$$\text{poly}\left(x^3 + 5\,x + 3, [x]\right)$$
```
>> op(polynomial, 1) = expr(polynomial)
```
$$x^3 + 5\,x + 3 = x^3 + 5\,x + 3$$

4.16.2 Computando com polinômios

A função **degree** determina o grau de um polinômio:

```
>> p := poly(1 + x + a*x^2*y, [x, y]):
>> degree(p, x), degree(p, y)
```
$$2,\ 1$$

Se você não especificar o nome de uma indeterminada como segundo argumento, então **degree** retorna o "grau total":

```
>> degree(p), degree(poly(x^27 + x + 1))
```
$$3,\ 27$$

A função **coeff** extrai coeficientes de um polinômio:

```
>> p := poly(1 + a*x + 7*x^7, [x]):
>> coeff(p, 1), coeff(p, 2), coeff(p, 8)
```
$$a,\ 0,\ 0$$

Para polinômios multivariados, o coeficiente de uma potência de um particular indeterminado é novamente um polinômio nos indeterminados restantes:

```
>> p := poly(1 + x + a*x^2*y, [x, y]):
>> coeff(p, y, 0), coeff(p, y, 1)
```
$$\text{poly}\left(x + 1, [x]\right),\ \text{poly}\left(a\,x^2, [x]\right)$$

Os operadores padrões +, -, *, e ^ trabalham para aritmética polinomial também:

```
>> p := poly(1 + a*x^2, [x]): q := poly(b + c*x, [x]):
>> p + q, p - q, p*q, p^2
          2
   poly(a x  + c x + (b + 1), [x]),

            2
      poly(a x  + (-c) x + (- b + 1), [x]),

                3          2
      poly((a c) x  + (a b) x  + c x + b, [x]),

           2  4          2
      poly(a  x  + (2 a) x  + 1, [x])
```

A função **divide** executa uma "divisão com resto":

```
>> p := poly(x^3 + 1): q := poly(x^2 - 1): divide(p, q)
```
$$\text{poly}\left(x, [x]\right),\ \text{poly}\left(x + 1, [x]\right)$$

O resultado é uma sequência com dois operandos: o quociente e o resto da divisão:

```
>> quotient  := op(divide(p, q), 1):
   remainder := op(divide(p, q), 2):
>> p = quotient*q + remainder
```
$$\text{poly}\left(x^3 + 1, [x]\right) = \text{poly}\left(x^3 + 1, [x]\right)$$

O polinômio denotado pelo **resto** é de grau menor que **q**, o qual faz a decomposição **p = quociente * q + resto** único. Dividindo dois polinômios pelos sentidos do operador usual / só é permitido no caso especial quando o resto que **divide** devolve desaparecendo:

```
>> p := poly(x^2 - 1): q := poly(x - 1): p/q
```
$$\text{poly}\left(x + 1, [x]\right)$$
```
>> p := poly(x^2 + 1): q := poly(x - 1): p/q
   FAIL
```

Note que os operadores aritméticos processam apenas polinômios de3 tipos exatamente iguais:

```
>> poly(x + y, [x, y]) + poly(x^2, [x, y]),
   poly(x) + poly(x, [x], Expr)
```

$$\text{poly}\left(x^2 + x + y, [x, y]\right),\ \text{poly}\left(2\,x, [x]\right)$$

Ambas as listas de indeterminados e o anel coeficiente devem coincidir, do contrário o sistema retorna a entrada como uma expressão simbólica:

```
>> poly(x + y, [x, y]) + poly(x^2, [x])
```

$$\text{poly}\left(x^2, [x]\right) + \text{poly}\left(x + y, [x, y]\right)$$

```
>> poly(x, Dom::Integer) + poly(x)
```

$$\text{poly}\left(x, [x], \mathbb{Z}\right) + \text{poly}\left(x, [x]\right)$$

O polinômio aritmético executa adições coeficientes e multiplicações de acordo com as regras do anel coeficiente:

```
>> p := poly(4*x + 11, [x], Dom::IntegerMod(3)):
>> p; p + p; p*p
```

$$\text{poly}\left(x + 2, [x], \mathbb{Z}_3\right)$$

$$\text{poly}\left(2\,x + 1, [x], \mathbb{Z}_3\right)$$

$$\text{poly}\left(x^2 + x + 1, [x], \mathbb{Z}_3\right)$$

O operador padrão ***** não trabalha imediatamente para multiplicar um polinômio por uma escalar; você precisa converter o fator escalar para um polinômio primeiro:

```
>> p := poly(x^2 + y):
>> scalar*p; poly(scalar, op(p, 2..3))*p
```

$$\text{scalar}\ \text{poly}\left(x^2 + y, [x, y]\right)$$

$$\text{poly}\left(\text{scalar}\ x^2 + \text{scalar}\ y, [x, y]\right)$$

Aqui nós nos certificamos que o polinômio gerado de um fator escalar é do mesmo tipo como p por passar op (p, 2..3) (=[x, y], Expr) como argumentos futuros para **poly**. Alternativamente, a função **multcoeffs** multiplica todos os coeficientes de um polinômio por um fator escalar:

```
>> multcoeffs(p, scalar)
```

$$\text{poly}\left(\text{scalar}\ x^2 + \text{scalar}\ y, [x, y]\right)$$

Estes rendimentos outro modo de multiplicação por uma escalar:

```
>> mapcoeffs(p, _mult, scalar)
```

$$\text{poly}\left((2\,\text{scalar})\ x^2 + (3\,\text{scalar})\ y, [x, y]\right)$$

Outra importante operação é a avaliação de um polinômio no ponto (computando um valor como imagem). A função **evalp** alcança isto:

```
>> p := poly(x^2 + 1, [x]):
   evalp(p, x = 2), evalp(p, x = x + y)
```

$$5,\ (x + y)^2 + 1$$

Esta computação é também válida para polinômios multivariados e rendimentos de um polinômio nas indeterminadas restantes ou, para um polinômio uni variado, um elemento do anel coeficiente:

```
>> p := poly(x^2 + y):
>> q := evalp(p, x = 0); evalp(q, y = 2)
```

$$\text{poly}\left(y, [y]\right)$$

$$2$$

Equivalentemente, você pode também considerar um polinômio como uma função das indeterminadas e chamar esta função com argumentos:

```
>> p(2, z)
```

$$z + 4$$

Uma variedade das funções do MuPAD aceita polinômios como entrada. Uma importante operação é a fatoração, a qual é interpretada de acordo com as regras do anel coeficiente. Você pode fazer fatoração com a função do MuPAD **factor**:

```
>> factor(poly(x^3 - 1))
```

$$\text{poly}\left(x - 1, [x]\right) \cdot \text{poly}\left(x^2 + x + 1, [x]\right)$$

```
>> factor(poly(x^2 + 1, Dom::IntegerMod(2)))
```

$$\text{poly}\left(x + 1, [x], \mathbb{Z}_2\right)^2$$

A função **D** diferencia polinômios:

```
>> D(poly(x^7 + x + 1))
```

$$\text{poly}\left(7\,x^6 + 1, [x]\right)$$

Equivalentemente, você pode usar também **diff (polynomial, x).**

Integração também trabalha para polinômios:

```
>> p := poly(x^7 + x + 1): int(p, x)
```

$$\text{poly}\left(\left(\frac{1}{8}\right)x^8 + \left(\frac{1}{2}\right)x^2 + x, [x]\right)$$

A função **gcd** computa o máximo divisor comum de polinômios:

```
>> p := poly((x + 1)^2*(x + 2)):
>> q := poly((x + 1)*(x + 2)^2):
>> factor(gcd(p, q))
```

$$\text{poly}(x+2,[x]) \cdot \text{poly}(x+1,[x])$$

A representação interna de um polinômio armazena somente estas potências das indeterminadas com coeficientes não desaparecendo. Isso é particularmente vantajoso para polinômios “escassos” de alto grau e faz aritmética com tais polinômios eficientes. A função **nterms** retorna o número de termos não triviais de um polinômio. A função **nthmonomial** extrai monômios individuais (tempos coeficientes potencias de indeterminados), **nthcoeff** e **nthterm** retorna o coeficiente apropriado e produto de potências de indeterminadas, respectivamente:

```
>> p := poly(a*x^100 + b*x^10 + c, [x]):
>> nterms(p), nthmonomial(p, 2),
   nthcoeff(p, 2), nthterm(p, 2)
```

$$3,\ \text{poly}\left(b\,x^{10},[x]\right),\ b,\ \text{poly}\left(x^{10},[x]\right)$$

`+, -, *, ^`	: arithmetic
`coeff`	: extract coefficients
`degree`	: polynomial degree
`diff, D`	: differentiation
`divide`	: division with remainder
`evalp`	: evaluation
`expr`	: conversion to expression
`factor`	: factorization
`gcd`	: greatest common divisor
`mapcoeffs`	: apply a function
`multcoeffs`	: multiplication by a scalar
`nterms`	: number of non-zero coefficients
`nthcoeff`	: n-th coefficient
`nthmonomial`	: n-th monomial
`nthterm`	: n-th term
`poly`	: construct a polynomial
`poly2list`	: conversion to list

Taberla 4.5: Funções do MuPAD operando em polinômios

Tabela 4.5 é um sumário das operações para polinômios discutidos acima. Seção “Funções para polinômios” do Guia rápido de referencia do MuPAD lista futuras funções para polinômios na biblioteca padrão. A biblioteca **groebner** cumpre funções para controlar polinômios ideais multivariados (veja **?groebner)**.

Exercício 4.42: Considere os polinômios $p = x^7 - x^4 + x^3 - 1$ e $q = x^3 - 1$. Compute $p - q^2$. q divide p? Fatore p e q.

Exercício 4.43: Um polinômio é chamado irredutível (acima de um campo coeficiente) se ele não pode ser fatorado dentro de um produto de mais que um polinômio não constante. A função **irreducible** testa um polinômio para irredutibilidade. Encontre todos os polinômios quadráticos irredutíveis $a\,x^2 + b\,x + c$, $a \neq 0$ além do campo das integrais de módulo 3.

4.17 Aritmética de intervalo

Com a versão do MuPAD 2.5, o novo tipo de kernel DOM_INTERVAL foi introduzido., Objetos deste tipo representam intervalos reais ou complexos de números de pontos flutuantes. Eles proveem, através de outras possibilidades, uns sentidos de controle um dos problemas mais fundamentais de aritmética dos pontos flutuantes: erros de arredondamentos.

A ideia básica é como se segue: Ao invés de pontos flutuantes números x_1, x_2 etc, onde a maioria de cada operação leva para erros (normalmente pequenos), considere intervalos X_1, X_2, etc. os quais são conhecidos por conter os números precisos vindos da aplicação. Um vai como ter uma alegação verificada que o valor $y = f(x_1, x_2,...)$ de uma função f pertence em algum intervalo
Y. Matematicamente, o conjunto imagem

$$Y = f(X_1, X_2, ...)\ \{f(\xi_1, \xi_2,...);\ \xi_1 \in X_1;\ \xi_2 \in X_2;\ldots\}$$

É desejado. Computando esse conjunto de imagem *exatamente* é uma tarefa formidável. De fato, isto é tão ambicioso para perguntar por uma representação exata quando usa memória rápida e eficiente de aritmética de pontos flutuantes. Ao invés disso, a versão intervalo de uma função f é um algoritmo $f^\wedge$ que produz um conjunto mais largo $f^\wedge(X_1, X_2,...)$ o qual é *garantido* para conter o jogo imagem exato de f:

$$f(X_1, X_2,...) \subset f^\wedge(X_1, X_2,...).$$

Se nós pensarmos os intervalos X_1, X_2 etc. como incorporação das inexatidões em x_1, x_2 etc. causada pelos arredondamentos, a versão intervalo $f^\wedge$ da função f provê os saltos altos e baixos do resultado de $y = f(x_1, x_2,...)$ no qual os erros de arredondamentos de x_1, x_2 etc. tem propagado. Alternativamente, você pode também considerar X_1,... como medidas imprecisas ou em algum outro caminho de quantidade indeterminada (“Este parâmetro está em algum lugar entre 0 e 1,” “este resistor tem uma tolerância de 10%”). Em tal configuração, o resultado $f^\wedge(X_1, X_2,...)$ é um conjunto *garantido* de conter todos os possíveis resultados.

Intervalos de pontos flutuantes são áreas retangulares no plano complexo consistindo de um intervalo para a parte real e um intervalo para a parte imaginária. Usando **hull** ou **...**, você pode especificar o retângulo por seus cantos “esquerda em baixo” e “direita acima”:

```
>> X3 := (2 - 3*I) ... (4 + 5*I)
    (2.0 ... 4.0) + i (−3.0 ... 5.0)
```

As funções do MuPAD que suportam intervalos aritméticos incluem as operações aritméticas básicas +, -, *, /, ^ tão bem como a maioria das funções especiais tais como **sin, cos, exp, ln, abs** etc.:

```
>> X1^2 + X2
    8.869604401 ... 9.695956224
>> X1 - I*X2 + X3
    (5.141592653 ... 7.141592654) + i (−2.826351823 ... 6.0)
>> sin(X1) + exp(abs(X3))
    7.389056098 ... 148.4131592
```

Quando dividido por um intervalo contendo 0, infinitos intervalos são produzidos. Os objetos **RD_INF** e **RD_NINF** (“arredondado infinito” e “arredondado infinito negativo,” respectivamente) representa os valores ±∞ em um contexto de intervalo:

```
>> sin(X2^2 - 1/2)
    −0.4527492553 ... 0.4794255387
>> 1/%
    RD_NINF ... − 2.208728094 ∪ 2.085829642 ... RD_INF
```

O último exemplo mostra que a aritmética pode produzir uniões “simbólicas” de intervalos de pontos flutuantes. Atualmente, a união é ainda do tipo **DOM_INTERVAL:**

```
>> domtype(%)
    DOM_INTERVAL
```

De fato, as funções **union** e **intersect** para criação de uniões e intersecções de conjuntos podem ser usadas para criar intervalos:

```
>> X1 union X2^2
    0.03015368960 ... 1.0 ∪ 3.141592653 ... 3.141592654
>> cos(X1*X2) intersect X2
    −1.0 ... − 0.1736481776
```

Objetos simbólicos tais como **identificadores** (Capítulo 4.3) e intervalos de pontos flutuantes podem ser misturados. A função **interval** substitui todas as subexpressões numéricas de uma **expression** (Capítulo 4.4) por intervalos de pontos flutuantes:

```
>> interval(2*x^2 + PI)
    (2.0 ... 2.0) x^2 + (3.141592653 ... 3.141592654)
```

No MuPAD, identificadores são assumidos implicitamente para representar valores complexos arbitrários. Consequentemente, a função **hull** substitui x pela reprsentação do intervalo no complexo plano inteiro:

```
>> hull(%)
    (RD_NINF ... RD_INF) + i (RD_NINF ... RD_INF)
```

Há um número de funções especializadas para intervalos de pontos flutuantes fixos como métodos do domínio kernel **DOM_INTERVAL**. Aqui, nós apenas mencionamos DOM_INTERVAL::center (o centro de um intervalo ou uma união de intervalos) e DOM_INTERVAL::width (a largura de um intervalo ou uma união de intervalos):

```
>> DOM_INTERVAL::center(2 ... 5 union 7 ... 9)
    5.5
>> DOM_INTERVAL::width(2 ... 5 union 7 ... 9)
    7.0
```

Veja **?DOM_INTERVAL** para um guia de sobrevivência dosa métodos disponíveis.

Futuramente, um domínio de biblioteca **Dom::FloatIV** o qual existe é somente um desvanecer para os intervalos de pontos flutuantes para o tipo do kernel **DOM_INTERVAL**. Isso é útil para embutir intervalos de pontos flutuantes em “containers” tais como matrizes (Capítulo 4.15) ou polinômios (Capítulo 4.16). Estes containers requerem a especificação do domínio do coeficiente que tem as propriedades matemáticas de um anel. Na ordem para fazer intervalos de pontos flutuantes embutidos em tais containers, o domínio **Dom::FloatIV** deu uma variedade de atributos matemáticos (“**categories”)** requeridas para tais propósitos:

```
>> Dom::FloatIV::allCategories()
   [Cat::Field, ...,  Cat::Ring, ...]
```

Em particular, o conjunto de todos os intervalos de ponto flutuante não é somente considerado como um anel, mas até mesmo como um campo. Estritamente falando, contudo, essas categorias matemáticas não são realmente adequadas para intervalos de objetos. Por exemplo, subtraindo um intervalo de si mesmo não faz render o elemento zero (o elemento neutro com respeito a adição):

```
>> (2 ... 3)  - (2 ... 3)
    -1.0 ... 1.0
```

Pragmaticamente, contudo, as categorias matemáticas foram conjunto, de qualquer modo, para habilitar embutindo. Por exemplo, nós consideramos o inverso de uma matriz de Hilbert (também veja Exercício 4.38). Estas matrizes são notoriamente incondicionadas. Para a matriz de Hilbert 8 X 8, o número condição (a razão do mais largo e do mais pequeno autovalor) é

```
>> A := linalg::hilbert(8):
   ev := numeric::eigenvalues(A):
   max(op(ev))/min(op(ev))
    15257583501.0
```

Asperamente falando, quando invertendo esta matriz por um algoritmo de ponto flutuante, um deve esperar perder por volta de 10 dígitos decimais de precisão:

```
>> log(10, %)
    10.18348576
```

Consequentemente, com baixos valores do comando DIGITS, o inverso deve ser substancialmente arruinado por arredondamento. Ao invés disso, depois da conversão de A para uma matriz de intervalos de pontos flutuantes, nós podemos usar o algoritmo genérico para uma inversão de matriz:

```
>> B := 1/Dom::Matrix(Dom::FloatIV)(A)
        array(1..8, 1..8,
          (1, 1) = -73.29144677 ... 201.2914468,
                ...
          (3, 2) = -955198.1290 ... -949921.8709,
                ...
          (8, 8) = 176679046.2 ... 176679673.8
        )
```

As entradas do inverso são *garantidas* de pertencer nos intervalos indicados. O componente (8,8) é determinado por uma precisão de 6 levando dígitos decimais, ainda o componente -73.29 e 201.3. Note, contudo, que a inversão genérica do algoritmo tende a superestimar os intervalos drasticamente. Os resultados retornados por uma inversão numérica com números pontos flutuante “padrão” *devem* atualmente ser mais precisos por este intervalo de computação.

Os componentes exatos do inverso estão disponíveis no MuPAD, também. Todas as entradas das matrizes inversas de Hilbert são integrais:

```
>> C := linalg::invhilbert(8)
                  array(1..8, 1..8,
                    (1, 1) = 64,
                           ...
                    (3, 2) = -952560,
                           ...
                    (8, 8) = 176679360
                  )
```

Usando o operador **in**, é possível determinar se um número ou expressão está dentro de um intervalo. Para combinar as matrizes, nós usamos **zip** (Capítulo 4.15.2) e verificamos para cada entrada do inverso exato de C se ele está dentro do intervalo na matriz B. Para este fim, a seguinte entrada converte ambas as matrizes dentro de listas de suas entradas e então executa a conferência:

```
>> zip([op(C)], [op(B)], (c, b) -> bool(c in b))
          [TRUE, TRUE, TRUE, TRUE, ..., TRUE]
```

4.18 **Objetos nulos:** null (), NIL, FAIL, undefined

Há vários objetos representando o “vazio” no MuPAD. Preimeiro, nós temos a “sequência vazia” gerada por **null** ().Este é do tipo de domínio **DOM_NULL** e não gera nenhuma saída na tela. Sistemas de funções tais como **reset** (Seção 14.3) ou **print** (Seção 13.1.1), as quais não podem retornar matematicamente valores úteis, retorna este objeto do MuPAD ao invés disso:

```
>> a := reset(): b := print("hello"):
                          "hello"

>> domtype(a), domtype(b)
    DOM_NULL, DOM_NULL
```

O objeto **null** () é particularmente útil em conexão com sequencias (Seção 4.5). O sistema remove automaticamente este objeto das sequências, e você pode usá-lo, por exemplo, para remover entradas de sequencia seletivamente:

```
>> delete a, b, c:
   Seq := a, b, c: Seq := eval(subs(Seq, b = null()))
    a, c
```

Aqui nós usamos o comando de substituição **subs** (capítulo 6) para substituir b pelo **null** ().

O objeto do MuPAD **NIL**, o qual é distinto de **null ()**, intuitivamente significa “sem valor”. Alguns sistemas de funções retornam o objeto **NIL** quando você chama eles com argumentos para cada eles não precisam computar nada. Um exemplo típico é a função **_if**, a qual é usualmente chamada na forma de uma declaração **if** (Capítulo 17):

```
>> condition := FALSE: if condition then x := 1 end_if
    NIL
```

Variáveis locais não inicializadas e parâmetros do MuPAD de procedimentos também tem o valor **NIL** (Seção 18.4).

O objeto do MuPAD **FAIL** intuitivamente significa “eu não posso encontrar um valor.” Sistemas de funções retornam esse objeto quando não há resultado si9gnificativo para um dado parâmetro de entrada. No seguinte exemplo, nós tentamos computar o inverso de uma matriz singular:

```
>> A := matrix([[1, 2], [2, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

```
>> A^(-1)
   FAIL
```

Outro objeto com um significado singular é **undefined**. Por exemplo, a função do MuPAD **limit** retorna este objeto quando o limite requisitado não existe:

```
>> limit(1/x, x = 0)
   undefined
```

Operações aritméticas com **undefined** retornam novamente **undefined**:

```
>> undefined + 1, 2^undefined
   undefined, undefined
```

Capítulo 5

Avaliação e simplificação

5.1 Identificadores e seus valores

Considere:

```
>> delete x, a: y := a + x
```

$$a + x$$

Desde que os identificadores **a** e **x** apenas representam a si mesmos, o “valor” de **y** é a expressão simbólica $a + x$. Nós temos que distinguir claramente entre o identificador **y** e seu valor. Mais precisamente, o *valor* de um identificador denota o mobjeto do MuPAD que o sistema computa pela avaliação e simplificação do lado de mão direita da declaração **identifier:=value** *no tempo da declaração.*

Note que no exemplo acima, o valor de **y** é composto dos identificadores simbólicos **a** e **x**, os quais podem ser nomeados valores em um tempo futuro. Por exemplo, se nós nomeamos o valor 1 para o identificador **a**, então **a** é substituído por seu valor 1 na expressão **a + x,** e a chamada **y** retorna $x + 1$:

```
>> a := 1: y
```

$$x + 1$$

Nós dizemos que a *avaliação* do identificador **y** retorna o resultado $x + 1$, mas seu *valor* é ainda **a + x:**
Nós distinguimos entre um identificador, seu valor, e sua avaliação: o *valor* denota a avaliação *no tempo da declaração*, uma *avaliação* futura pode retornar um *“valor corrente”* diferente.

Se nós agora nomeamos o valor 2 para **x**, então ambos **a** e **x** são substituídos por seus valores na próxima avaliação de **y**. Desde que nós obtemos a soma 2 + 1 como um resultado, do qual o MuPAD automaticamente simplifica para 3:

```
>> x := 2: y
   3
```

A *avaliação* de **y** agora retorna o número 3; seu *valor* é ainda $a + x$.

Isso é razoável para dizer que o valor de **y** é o resultado no tempo da tarefa. Isto é, se nós apagarmos os valores dos identificadores **a** e **x** no exemplo acima, então a avaliação de **y** rende seu valor original imediatamente depois da tarefa:

```
>> delete a, x: y
```

$$a + x$$

Se **a** ou **x** já tem um valor *antes de* nós nomearmos a expressão **a + x** para **y**, então acontece o seguinte:

```
>> x := 1: y := a + x: y
```

$$a + 1$$

No tempo da tarefa, **y** é nomeado a avaliação de **a + x**, por exemplo, $a + 1$. Ao invés disso, este é agora o *valor* de **y**, o qual não contém referência para **x**:

```
>> delete x: y
```

$$a + 1$$

Aqui ao alguns exemplos futuros para este mecanismo. Nós primeiro nomeamos o número racional **1/3** para **x**, então nós nomeamos os objetos **[x, x^2, x^3]** para o identificador **list**. Na nomeação, o sistema avalia o lado da mão direita e automaticamente substitui o identificador **x** por seu valor. Assim, no tempo da nomeação, o identificador **lista** consegue o valor $[1/3, 1/9, 1/27]$ e não $[x, x^2, x^3]$:

```
>> x := 1/3: list := [x, x^2, x^3]
```

$$\left[\frac{1}{3}, \frac{1}{9}, \frac{1}{27}\right]$$

```
>> delete x: list
```

$$\left[\frac{1}{3}, \frac{1}{9}, \frac{1}{27}\right]$$

MuPAD aplica o mesmo esquema de avaliação para chamadas de funções simbólicas:

```
>> delete f: y := f(PI)
```

$$f(\pi)$$

Depois da nomeação

```
>> f := sin:
```

Nós obtemos a avaliação

```
>> y
```

$$0$$

Quando avaliando **y**, o sistema substitui o identificador **f** por seu valor, o qual é o valor do identificador **sin.** Esse é um procedimento do qual é executado quando **y** é avaliado e retorna sin (π) como 0.

5.2 Completa, Incompleta, e Avaliação Forçada.

Nós consideramos, mais uma vez o primeiro exemplo da seção anterior. Ali nós nomeamos a expressão **a + x** para o identificador **y**, e **a** e **x** não tem um valor:

```
>> delete a, x: y := a + x: a := 1: y
```

$$x + 1$$

Nós agora explicamos em maiores detalhes como o MuPAD executa a avaliação final.

Primeiro (“nível 1”) o avaliador considera o valor *a* + *x* de **y**. Desde que este valor contém o identificador **x** e **a**, um segundo passo (“nível 2”) é necessário para determinar o valor desses identificadores. O sistema reconhece que **a** tem o valor 1, enquanto **x** não tem valor (e assim matematicamente representa uma incógnita). Agora a aritmética do sistema combina esses resultados para *x* + 1, e essa é a avaliação de **y**. Figuras 5.1 – 5.3 ilustram esse processo. Uma caixa representa num identificador e seu valor (ou ., respectivamente, se ele não tem valor). Uma seta representa um passo de avaliação.

Figura 5.1: O identificador **y** sem um valor

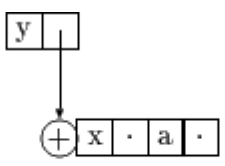


Figura 5.2: Após a nomeação **y:=a + x.**

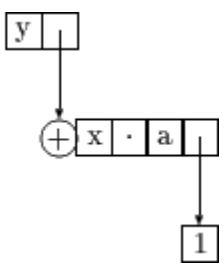


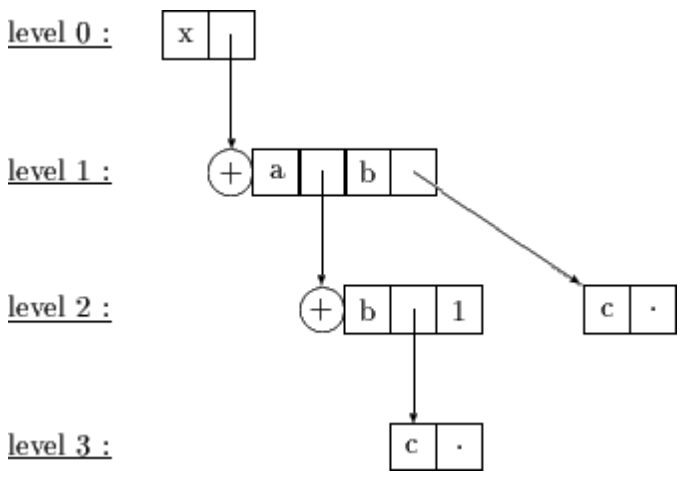
Figura 5.3: Após a nomeação **a:=1**, nós finalmente obtemos **x + 1** como a avaliação de **y.**

Em analogia as árvores de expressão para representar expressões simbólicas (Seção 4.4.2), você pode imaginar o processo de avaliação como uma *árvore de avaliação*, a qual vértices são expressões com identificadores simbólicos, com filiais apontando para os valores correspondendo desses identificadores. O sistema atravessa esta arvore até haver qualquer um nenhum identificador à esquerda ou todos os identificadores restantes não tiverem nenhum valor.

O usuário pode controlar os níveis desta árvore via função do sistema **level.** Nós olhamos no exemplo:

```
>> delete a, b, c: x := a + b: a := b + 1: b := c:
```

A árvore de avaliação para **x** é:



O identificador **x** forma o nível do topo (a raiz, nível 0) de sua própria árvore de avaliação:

```
>> level(x, 0)
```

$$x$$

O próximo nível 1 determina o valor de **x**:

```
>> level(x, 1)
    a + b
```

No seguinte nível 2, **a** e **b** são substituídos pelos seus valores $b + 1$ e c, respectivamente:

```
>> level(x, 2)
    b + c + 1
```

O **b** restante é substituído por seu valor *c* apenas no próximo nível 3:

```
>> level(x, 3)
    2 c + 1
```

Nós chamamos o tipo de avaliação descrito aqui uma *avaliação completa*. Isso significa que os identificadores são substituídos por seus valores recursivamente até não serem possíveis futuras avaliações. A variável ambiente **LEVEL**, a qual tem um valor padrão 100, determina quão distante MuPAD desce na maioria das árvores de avaliação.

No modo interativo, MuPAD sempre avalia acima para a profundidade **LEVEL** no modo interativo.

```
>> delete a0, a1, a2: LEVEL := 2:
>> a0 := a1: a0
    a1
>> a1 := a2: a0
    a2
```

Acima para agora, a árvore de avaliação para **a0** tem profundidade 2, e o valor **LEVEL** de 2 fixa um a avaliação completa.

Contudo, no próximo passo,m o valor de **a2** não é levado em conta:

```
>> a2 := a3: a0
    a2
>> delete LEVEL:
```

Tão cedo quanto MuPAD realiza que o nível de avaliação corrente excede o valor da variável ambiente **MAXLEVEL** (pela qual o valor padrão é 100), então isso assume para estar em um infinito loop e aborta a avaliação com uma mensagem de erro:

```
>> MAXLEVEL := 2: a0
   Error: Recursive definition [See ?MAXLEVEL]

>> delete MAXLEVEL:
```

Nós agora apresentamos algumas exceções importantes para a regra da avaliação completa. A chamada **last (i), %1, ou %** (Capítulo 12) não leva para a avaliação! Nós consideramos o exemplo

```
>> delete x: [sin(x), cos(x)]: x := 0:
```

Agora, **%2** acessa a lista sem avaliá-la:

```
>> eval(%)
    [0, 1]
```

Compare isto para as seguintes declarações, onde requerem o identificador **list** causa a avaliação usual completa:

```
>> delete x: list := [sin(x), cos(x)]: x := 0: list
    [0, 1]
```

Arranjos do tipo de domínio **DOM_ARRAY** são sempre avaliados com o nível 1:

```
>> delete a, b: A := array(1..2, [a, b]):
>> b := a: a := 1: A
    ( a  b )
```

Como você pode ver, a chamada de **A** retorna o valor (o arranjo), mas não substitui **a, b** por seus valores. Você pode avaliar as entradas via **map (A, eval):**

```
>> map(A, eval)
    ( 1  1 )
```

Note que em contraste com o comportamento acima, o acesso indexado de uma entrada individual é avaliada completamente:

```
>> A[1], A[2]
    1, 1
```

Matrizes (do tipo **Dom::Matrix(.))**, tabelas (do tipo de domínio **DOM_TABLE)**, e polinômios (**DOM_POLY)** são tratados no mesmo modo de arranjos. Ainda mais, sem procedimentos, MuPAD sempre avalia somente acima para nível 1 (Seção 18.11). Se isso não é suficiente, você pode controlar este comportamento explicitamente pelos sentidos de **level.**

O comando **hold (object)** é similar a **level (object,0)** e previne a avaliação de **object**. Isso pode ser desejável em várias situações. A seguinte função, da qual não pode ser executada para argumentos simbólicos, rende um exemplo onde a (prematura) avaliação é indesejável:

```
>> absValue := X -> (if X >= 0 then X else -X end_if):
>> absValue(X)
   Error: Can't evaluate to boolean [_leequal];
   during evaluation of 'absValue'
```

Se você quer integrar numericamente esta função usando **numeric::int**, a entrada óbvia retorna o mesmo erro:

```
>> numeric::int(absValue(X), X = -1..1)
```

Se você demora a avaliação de **absValue (X)** por **hold**, o valor é computado:

```
>> numeric::int(hold(absValue)(X), X = -1..1)
    1.0
```

A razão é que **numeric::int** substitui internamente valores numéricos para X, para qual **absValue** pode ser avaliado sem problemas.

Aqui está outro exemplo: como a maioria das funções do MuPAD, a função **domtype** primeiro avalia seu argumento, assim que o comando **domtype (object)** retorna o tipo de domínio da *avaliação* do objeto.

```
>> x := 1: y := 1: x, x + y, sin(0), sin(0.1)
    1, 2, 0, 0.09983341665
>> domtype(x), domtype(x + y), domtype(sin(0)),
   domtype(sin(0.1))
    DOM_INT, DOM_INT, DOM_INT, DOM_FLOAT
```

Usando **hold**, você obtêm os tipos de domínio dos objetos por si mesmos: **x** é um identificador, **x + y** é uma expressão, e sin (0) e sin (0.1) são chamadas de funções e consequentemente expressões tão bem:

```
>> domtype(hold(x)), domtype(hold(x + y)),
   domtype(hold(sin(0))), domtype(hold(sin(0.1)))
    DOM_IDENT, DOM_EXPR, DOM_EXPR, DOM_EXPR
```

O comando **?level** e **?hold** proveem informação futura das páginas de ajuda correspondentes.

Exercício 5.1: Quais são os *valores* dos identificadores **x, y,** e **z** depois das seguintes afirmações em cada caso?

```
>> delete a1, b1, c1, x:
   x := a1: a1 := b1: a1 := c1: x
>> delete a2, b2, c2, y:
   a2 := b2: y := a2: b2 := c2: y
>> delete a3, b3, z:
   b3 := a3: z := b3: a3 := 10: z
```

Prediga os resultados das seguintes sequências afirmadas:

```
>> delete u1, v1, w1:
   u1 := v1: v1 := w1: w1 := u1: u1
>> delete u2, v2:
   u2 := v2: u2 := u2^2 - 1: u2
```

5.3 Simplificação Automática

MuPAD automaticamente simplifica muitos objetos tais como certa chamadas de função ou expressões aritméticas com números:

```
>> sin(15*PI), exp(0), (1 + I)*(1 - I)
    0, 1, 2
```

O mesmo possui verdadeiro para expressões aritméticas contendo **infinity:**

```
>> 2*infinity - 5
    ∞
```

Tais simplificações automáticas servem para reduzir a complexidade de expressões sem um requerimento explicito pelo usuário:

```
>> cos(1 + exp((-1)^(1/2)*PI))
    1
```

O usuário pode nem controlar nem estender a simplificação automática.
Na maioria dos casos, contudo, MuPAD *não* simplifica automaticamente expressões. A razão é que o sistema geralmente não pode decidir qual é o modo mais razoável de simplificação. Por exemplo, considere a seguinte expressão a qual não está simplificada:

```
>> y := (-4*x + x^2 + x^3 - 4)*(7*x - 5*x^2 + x^3 - 3)
    (−x^3 − x^2 + 4 x + 4) (−x^3 + 5 x^2 − 7 x + 3)
```

Naturalmente, você pode expandir esta expressão, que pode ser razoável, por exemplo, antes de computar sua integral simbólica:

```
>> expand(y)
    x^6 − 4 x^5 − 2 x^4 + 20 x^3 − 11 x^2 − 16 x + 12
```

Entretanto, se você está interessado nas raízes do polinômio, isso faz mais sentido para computar isso em fatores lineares:
Sem resposta universal é possível por qual questão de duas representações estar “simplificada.” Dependendo da aplicação pretendida, você pode seletivamente aplicar sistemas de funções apropriados (tais como **expando** ou **factor**) para simplificar uma expressão.

Há outro argumento contra simplificação automática. O símbolo *f*, por exemplo, pode representar uma função saltada, pela qual o limite $\lim_{x\to 0} x\, f(x)$ é 0. Entretanto, simplificando a expressão para 0 pode ser errado para funções *f* com uma singularidade na origem tais como $f(x) = 1/x$! Assim, simplificações automáticas tais como $0 \,.\, f(0) = 0$ são questionáveis tão longo como o

sistema não tem conhecimento adicional sobre o símbolo envolvido. Em geral, MuPAD não pode saber qual regra pode ser aplicada e qual não pode ser. Agora você poderia contestar que MuPAD não deve executar nenhuma simplificação em toso invés de uns errados. Infelizmente, isto não é razoável se, desde em computações simbólicas, expressões tendem a crescer muito rapidamente, e isto afeta seriamente a performance do sistema. De fato, na maioria dos casos MuPAD simplifica uma expressão da forma 0 * y para 0, exceto, por exemplo, quando o valor de **y** é **infinity**, **FAIL**, ou **undenied**. Você deve sempre guardar em mente que um tal resultado simplificado pode ser errado em casos extremos.

Por esta razão, MuPAD executa somente simplificações automaticamente, e você deve requerer explicitamente outras simplificações por si mesmas. Por este propósito MuPAD provê uma variedade de funções, algumas das quais são descritas na Seção 9.2.

Enquanto as simplificações automáticas foram cuidadosamente escolhidas, existem ainda exemplos onde eles levam a resultados inesperados, normalmente por conta das ambiguidades em notação matemática. Um exemplo é a solução da equação $x/x = 1$ *para* $x \neq 0$:

```
>> solve(x/x = 1, x)
    ℂ
```

O resoluto de equação do MuPAD **solve** (Capíktulo 8) retorna o conjunto $\mathbb{C}$ de todos os números complexos. Assim, **solve** clama que valores complexos *arbitrários* de x rende uma solução da equação $x/x = 1$. A razão é que o sistema primeiro simplifica a expressão **x/x** para 1, e então de fato resolve a equação 1 = 1. O caso excepcional $x = 0$, pelo qual o problema original não faz sentido, é completamente ignorado na saída simplificada!

Capítulo 6
Substituição: subs, subsex **e** subsop

Todos os objetos do MuPAD consistem de operandos (Seção 4.1). Uma participação importante de um sistema de álgebra computacional e que ela pode substituir estes blocos de construção por novos valores. Para este propósito, MuPAD provê as funções **subs, subsex** (atalho para: substitute expression), e **subsop** (atalho para: substitute operand).

O comando **subs (object, Old = New)** substituí todas as ocorrências da subexpressão Old em object pelo valor **New:**

```
>> f := a + b*c^b: g := subs(f, b = 2):  f, g
    a + b c^b, 2 c^2 + a
```

Você vê que **subs** retorna o resultado da substituição, mas o identificador **f** permanece inalterado. Se você representa um mapa F pela expressão $f = F(x)$, então você pode usar **subs** para avaliar a função em algum ponto:

```
>> f := 1 + x + x^2:
>> subs(f, x = 0), subs(f, x = 1),
   subs(f, x = 2), subs(f, x = 3)
    1, 3, 7, 13
```

A saída do comando **subs** é sujeitada as simplificações usuais do simplificador interno do MuPAD. No exemplo acima, a chamada subs (f, x = 0) produz o objeto 1 + 0 + 0^2, pelo qual é automaticamente simplificado para 1. Você não deve confundir isto com *avaliação* (Capítulo 5), onde em adição todos os identificadores em uma expressão são recolocados por seus valores.

A função **subs** executa uma substituição. O sistema apenas simplifica o objeto resultante, mas não o avalia após retornar!

No seguinte exemplo

```
>> f := x + sin(x): g := subs(f, x = 0)
    sin(0)
```

O identificador **sin** para a função seno não é substituído pela função do MuPAD correspondente, a qual vai retornar sin (0) = 0. Apenas a próxima chamada para **g** executa uma avaliação completa:

```
>> g
    0
```

Você pode forçar avaliação pelo uso de **eval:**

```
>> eval(subs(f, x = 0))
    0
```

Você pode substituir objetos do MuPAD arbitrários pela substituição. Em particular, você pode substituir funções ou procedimentos com novos valores:

```
>> eval(subs(h(a + b), h = (x -> (1 + x^2))))
    (a + b)^2 + 1
```

Se você quer substituir um sistema de função, inclua seu nome em um comando **hold**:

```
>> eval(subs(sin(a + b), hold(sin) = (x -> x - x^3/3)))
    a + b - (a + b)^3/3
```

Você pode também substituir subexpressões mais complexas:

```
>> subs(sin(x)/(sin(x) + cos(x)), sin(x) + cos(x) = 1)
    sin(x)
```

Você deve ser cuidadoso com tais substituições: o comando **subs (object, Old = New)** substitui todas essas ocorrências da expressão **Old** *que pode ser encontrada pelos sentidos de* ***op.*** Isto explica como nada acontece no seguinte exemplo:

```
>> subs(a + b + c, a + b = 1), subs(a * b * c, a * b = 1)
```
$$a+b+c,\ a\,b\,c$$

Aqui a soma **a + b** e o produto **a * b** *não* são operandos das expressões correspondentes. Até pior, nós encontramos:

```
>> f := a + b + sin(a + b): subs(f, a + b = 1)
```
$$a+b+\sin(1)$$

Novamente, você não pode obter a subexpressão **a + b** da soma exterior pelos sentidos de **op.** Contudo, o argumento do seno é o suboperando **op (f, [3, 1])** (veja seções 4.1 e 4.4.3), e consequentemente ela é substituída por 1.

Em contraste com **subs**, a função **subsex** é muito mais lenta que **subs** para objetos largos. Quando substituindo subexpressões mais complexas, você nãoi deve ser enganado pela saída da tela de expressões:

```
>> f := a/(b*c)
```
$$\frac{a}{b\,c}$$
```
>> subs(f, b*c = New), subsex(f, b*c = New)
```
$$\frac{a}{b\,c},\ \frac{a}{b\,c}$$

Se você olhar nos operandos de **f**, você vê que a árvore de expressão não faz conter o produto **b*c**. Isso explica porque substituição não toma lugar:

```
>> op(f)
```
$$a,\ \frac{1}{b},\ \frac{1}{c}$$

Você pode executar várias substituições com uma única chamada de **subs:**

```
>> subs(a + b + c, a = A, b = B, c = C)
```
$$A+B+C$$

Isso é equivalente para a chamada aninhada

```
>> subs(subs(subs(a + b + c, a = A), b = B), c = C):
```

Assim nós obtemos:

```
>> subs(a + b^2, a = b, b = a)
```
$$a^2+a$$

Primeiro, MuPAD substitui **a** por **b**, rendendo **b = b^2.** Então ele substitui **a** por **b** nesta nova expressão e retorna o resultado acima. Em contraste, você pode alcançar uma *substituição simultânea* por especificar as equações substituição na forma de uma lista ou um conjunto:

```
>> subs(a + b^2, [a = b, b = a]),
   subs(a + b^2, {a = b, b = a})
```
$$a^2+b,\ a^2+b$$

A saída do resoluto de equação **solve** (Capítulo 8) suporta a funcionalidade de **subs.** Em geral, **solve** retorna uma lista de equações, as quais podem ser usadas em **subs:**

```
>> equations := {x + y = 2, x - y = 1}:
>> solution := solve(equations, {x, y})
```
$$\left\{\left[x=\frac{3}{2}, y=\frac{1}{2}\right]\right\}$$
```
>> subs(equations, op(solution, 1))
```
$$\{1=1, 2=2\}$$

A função **subsop** provê outra variante de substituição: **subsop (object, i = New)** substitui seletivamente o iº operando do objeto pelo valor **New:**

```
>> subsop(2*c + a^2, 2 = d^5)
```
$$d^5+2\,c$$

Aqui, nós substituímos o segundo operando **a^2** da soma por **d^5.** No seguinte exemplo, nós primeiro substituímos o expoente do segundo termo (este é o operando [2, 2] da soma), e então o primeiro termo:

```
>> subsop(2*c + a^2, [2, 2] = 4, 1 = x*y)
```
$$a^4+x\,y$$

Na seguinte expressão, nós primeiro substituímos o primeiro termo, rendendo a expressão **x * y + c^2**. Então nós substituímos **z** pelo segundo fator do primeiro termo (que agora é **y**):

```
>> subsop(a*b + c^2, 1 = x*y, [1, 2] = z)
```
$$c^2+x\,z$$

A expressão **a + 2** é uma soma simbólica, da qual tem um 0º operando, chamado, a função do sistema **_plus** para gerar somas:

```
>> op(a + 2, 0)
    _plus
```

Você pode substituir esse operando por qualquer outra função (por exemplo, pela função do sistema **_mult** que multiplica esses argumentos):

```
>> subsop(a + 2, 0 = _mult)
    2 a
```

Quando usando **subsop**, você precisa conhecer a posição do operando que você quer substituir. No entanto, você deve ser cauteloso, desde que o sistema pode mudar a ordem dos operandos quando este é matematicamente válido (por exemplo, em somas, produtos, ou conjuntos):

```
>> set := {sin(1 + a), a, b, c^2}
    {a, b, sin(a + 1), c^2}
```

Se você usa **subs**, você não precisa conhecer a posição da subexpressão. Outra diferença entre **subs** e **subsop** é que **subs** atravessa a árvore de expressão do objeto *recursivamente*, e assim também substitui suboperandos:

```
>> subs(set, a = a^2)
    {b, sin(a^2 + 1), a^2, c^2}
```

Exercício 6.1: Como o comando **subsop (b+ a, 1 = c)** substitui o identificador **b** por **c**?

Exercício 6.2: Os comandos

```
>> delete f: g := diff(f(x)/diff(f(x), x), x $ 5)
   25 diff(f(x), x, x) diff(f(x), x, x, x, x)
   ------------------------------------------ -
                                2
                 diff(f(x), x)

     4 diff(f(x), x, x, x, x, x)
     --------------------------- - ...
           diff(f(x), x)
```

geram uma expressão prolongada contendo derivadas simbólicas. Faça essa expressão mais legível substituindo estas derivadas pelos nomes simplificados $f_0 = f(x)$, $f_1 = f'(x)$, etc.

Capítulo 7
Diferenciação e Integração

Nós já usamos os comandos do MuPAD para diferenciação e integração. Desde que eles são importantes, nós recapitulamos o uso dessas rotinas aqui.

7.1 Diferenciação

A chamada **diff** (**expression, x)** computa a derivada das expressões com respeito a incógnita **x**:

```
>> diff(sin(x^2), x)
    2 x cos(x^2)
```

Se a expressão contém chamadas simbólicas para funções as quais derivadas não são conhecidas, então **diff** retorna por si mesma simbolicamente:

```
>> diff(x*f(x), x)
    f(x) + x ∂/∂x f(x)
```

Você pode computar altas derivadas via **diff (expression, x, x,...).** A sequência **x, x,...** de identificadores pode ser gerada convenientemente via o operador sequência $ (Seção 4.5):

```
>> diff(sin(x^2), x, x, x) = diff(sin(x^2), x $ 3)
    -12 x sin(x^2) - 8 x^3 cos(x^2) = -12 x sin(x^2) - 8 x^3 cos(x^2)
```

Você pode computar derivadas parciais do mesmo modo. MuPAD assume que derivadas parciais mistas de expressões simbólicas são simétricas:

```
>> diff(f(x,y), x, y) - diff(f(x,y), y, x)
    0
```

Se um mapa matemático é representado por uma função ao invés de uma expressão, então o operador diferencial **D** computa a derivada como uma função:

```
>> D(sin), D(exp), D(ln), D(sin*cos), D(sin@ln), D(f+g)
    cos, exp, 1/id, cos^2 - sin^2, (cos ∘ ln)/id, f' + g'
>> f := x -> (sin(ln(x))): D(f)
    x ↦ cos(ln(x))/x
```

Aqui, **id** denota o mapa identidade $x \mapsto x$. A expressão **D (f) (x)** retorna o valor da derivada em um ponto:

```
>> D(f)(1), D(f)(y^2), D(g)(0)
```

$$1,\ \frac{\cos\left(\ln\left(y^2\right)\right)}{y^2},\ g'(0)$$

O sistema converte o primo ‘ para a derivada para uma chamada de **D:**

```
>> f'(1), f'(y^2), g'(0)
```

$$1,\ \frac{\cos\left(\ln\left(y^2\right)\right)}{y^2},\ g'(0)$$

Para uma função com mais que um argumento, D ([i], f) é uma derivada parcial com respeito ao iº argumento, e D ([i, j,...], f) é equivalente a D ([i], D ([j],...)), para altas derivadas parciais.

Exercício 7.1: considere a função f: $x \rightarrow \sin(x)/x$. Compute primeiro o valor de f no ponto $x = 1.23$, e então a derivada $f'(x)$. Como faz a seguinte entrada não render o resultado desejado?

```
>> f := sin(x)/x: x := 1.23: diff(f, x)
```

Exercício 7.2: A regra de D l’Hospital declara que

$$\lim_{x\to x_0}\frac{f(x)}{g(x)} = \lim_{x\to x_0}\frac{f'(x)}{g'(x)} = \cdots = \lim_{x\to x_0}\frac{f^{(k-1)}(x)}{g^{(k-1)}(x)} = \frac{f^{(k)}(x_0)}{g^{(k)}(x_0)},$$

$$\text{if } f(x_0) = g(x_0) = \cdots = f^{(k-1)}(x_0) = g^{(k-1)}(x_0) = 0 \text{ and } g^{(k)}(x_0) \neq 0.$$

Compute $\lim_{x\to 0}\frac{x^3\sin(x)}{(1-\cos(x))^2}$ aplicando essa regra interativamente. Use a função **limit** para checar seu resultado.

Exercício 7.3: Determine a primeira e a segunda ordem de derivadas parciais de $f_1(x_1, x_2) = \sin(x_1, x_2)$. Deixe $x = x(t) = \sin(t)$, $y = y(t) = \cos(t)$, *e* $f_2(x, y) = x^2\ y^2$. Compute a derivada de $f_2\ (x(t), y(t))$ com respeito a t.

7.2 Integração

A função **int** caracteriza ambas integração definida e indefinida:

```
>> int(sin(x), x), int(sin(x), x = 0..PI/2)
```

$$-\cos(x),\ 1$$

Se **int** é incapaz de computar um resultado, isso retorna por si mesmo simbolicamente. No seguinte exemplo, o integrando é dividido dentro de dois termos. Apenas um destes tem uma integral que pode ser representada por funções elementares:

```
>> int((x - 1)/(x*sqrt(x^3 + 1)), x)
```

$$\frac{\ln\left(\frac{\left(\sqrt{x^3+1}-1\right)\left(\sqrt{x^3+1}+1\right)^3}{x^6}\right)}{3} - \int -\frac{1}{\sqrt{x^3+1}}\,\mathrm{d}x$$

A função $\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$ é implementado como uma função especial no MuPAD:

```
>> int(exp(-a*x^2), x)
```

$$\frac{\sqrt{\pi}\ \mathrm{erf}\left(\sqrt{a}\,x\right)}{2\sqrt{a}}$$

Enquanto integra, a variável de integração por si só é assumida para ser real. Outra como esta, todas as computações tomam lugar acima dos números complexos, e o sistema considera todo parâmetro simbólico em integrando como um número complexo a menos que contasse caso contrário. No seguinte exemplo, a integral definida existe apenas quando o parâmetro a tem uma parte real positiva, e o sistema retorna uma chamada simbólica **int:**

```
>> int(exp(-a*x^2), x = 0..infinity)
```

$$\int_0^\infty \mathrm{e}^{-a\,x^2}\,\mathrm{d}x$$

Você pode dizer ao sistema que um identificador tem certas propriedades pelo uso da função **assume** (Seção 9.3). A seguinte chamada para **assume** estipula que a ser um número real positivo:

```
>> assume(a > 0):
   int(exp(-a*x^2), x = 0..infinity)
```

$$\frac{\sqrt{\pi}}{2\sqrt{a}}$$

Além da computação exata de integrais definidas, MuPAD também provê vários métodos numéricos:

```
>> float(int(exp(-x^2), x = 0..2))
```

$$0.8820813908$$

Na computação prévia, **int** primeiro retorna um resultado simbólico (a função erf), a qual é então aproximada por **float.** Se você quer computar numericamente do início, então você pode suprimir a computação simbólica via **int** usando **hold** (Seção 5.2):

```
>> float(hold(int)(exp(-x^2), x = 0..2))
```

$$0.8820813908$$

Alternativamente, a função **numeric::int** da biblioteca **numeric** pode ser usada:

```
>> numeric::int(exp(-x^2), x = 0..2)
    0.8820813908
```

Essa função permite a você escolher diferentes métodos numéricos para computar a integral. Você encontra informação mais detalhada com **?numeric::int.** Isto trabalha em um estilo puramente numérico sem qualquer pré processamento simbólico do integrando. Para suavizar integrandos sem singularidades, **numeric::int** é totalemente eficiente.

Exercício 7.4: Compute as seguintes integrais:

$$\int_0^{\pi/2} \sin(x)\,\cos(x)\,\mathrm{d}x\,, \quad \int_0^1 \frac{\mathrm{d}x}{\sqrt{1-x^2}}\,, \quad \int_0^1 x\,\arctan(x)\,\mathrm{d}x\,.$$

Use MuPAD para verificar a seguinte igualdade: $\int_{-2}^{} \frac{}{x} = -\ln(2)$.

Exercício 7.5: Use MuPAD para determinar as seguintes integrais indefinidas:

$$\int^x \frac{t\,\mathrm{d}t}{\sqrt{(2\,a\,t-t^2)^3}}\,, \quad \int^x \sqrt{t^2-a^2}\,\mathrm{d}t\,, \quad \int^x \frac{\mathrm{d}t}{t\,\sqrt{1+t^2}}\,.$$

Exercício 7.6: A função **intlib::changevar** executa uma mudança de variável em uma integral simbólica. Leia a página de ajuda correspondente. MuPAD não pode computar a integral

$$\int_{-\pi/2}^{\pi/2} \sin(x)\,\sqrt{1+\sin(x)}\,\mathrm{d}x.$$

Ajude o sistema usando a substituição **t = sin(x).** Compare o valor que você obt~em para o resultado numérico retornado pela função **numeric::int.**

Capítulo 8

Resolvendo Equações: solve

A função **solve** resolve sistemas de equações. Esta rotina pode controlar vários tipos diferentes de equações. Além de equações “algébricas”, também certas classes de diferenciais e recorrentes equações podem ser resolvidas. Futuramente, existem uma variedade de resolvedores que apenas mantém classes especiais de equações. Muitas desses algoritmos são chamados atualmente por “universal” **solve** uma vez isso tem identificado o tipo da equação. O usuário pode também chamar os resolvedores especializados diretamente. Com **?solvers**, você pode requisitar um guia de sobrevivência de todos os solucionadores que estão disponíveis no MuPAD.

8.1 Equações Polinomiais

Você pode fornecer uma equação individual como primeiro argumento para **solve.** A incógnita pela qual você quer resolver é o segundo argumento:

```
>> solve(x^2 + x = y/4, x), solve(x^2 + x - y/4 = 0, y)
```

$$\left\{-\frac{\sqrt{y+1}}{2}-\frac{1}{2}, \frac{\sqrt{y+1}}{2}-\frac{1}{2}\right\}, \ \{4\,x^2+4\,x\}$$

Nesse caso, o sistema retorna um conjunto de soluções. Se você especificar uma expressão ao invés de uma equação, então **solve** assume a equação **expression=0:**

```
>> solve(x^2 + x - y/4, y)
```

$$\{4\,x^2+4\,x\}$$

Para polinômios de alto grau, isso provavelmente é impossível de sempre encontrar uma formula fechada para a solução por meio de radicais etc. Em tais casos, MuPAD usa o objeto **RootOf:**

```
>> solve(x^7 + x^2 + x, x)
```

$$\{0\} \cup \mathrm{RootOf}(\mathrm{X1}^6+\mathrm{X1}+1, \mathrm{X1})$$

O objeto **RootOf** acima representa todas as soluções da equação **x^6 +x +1 = 0.** Você pode usar **float** para aproximar tais objetos pelos números de pontos flutuantes. O sistema emprega internamente um procedimento numérico para determinar todas as (complexas) raízes do polinômio:

```
>> map(%, float)
   {0.0, 0.9454023333 + 0.6118366938 I,

      0.9454023333 - 0.6118366938 I,

      - 0.1547351445 + 1.038380754 I,

      - 0.1547351445 - 1.038380754 I,

      - 0.7906671888 - 0.3005069203 I,

      - 0.7906671888 + 0.3005069203 I}
```

Se você quer resolver uma colação de equações para possivelmente várias incógnitas, especifique ambas as equações e as incógnitas como conjuntos. No seguinte exemplo, nós resolvemos duas equações *lineares* em três incógnitas:

```
>> delete x, y, z:
   equations := {x + y + z = 3, x + y = 2}:
>> solution := solve(equations, {x, y, z})
```

$$\{[x = 2 - y, z = 1]\}$$

Se você resolve equações para várias variáveis, então MuPAD retorna um conjunto de "equações resolvidas" equivalente ao sistema original de equações. Você pode agora ler em voz alta as soluções imediatamente: a incógnita *z* tem o valor 1, a incógnita *y* pode ser arbitrária e, para qualquer valor dado de *y*, nós temos $x = 2 - y$. A chamada para **solve** não nomeia os valores para as incógnitas; *x* e *z* ainda são incógnitas. Contudo, a forma da saída como uma lista de equações resolvidas é escolhida de um tal modo que você pode convenientemente usar **subs** (Capítulo 6) para substituir esses valores em outros objetos. Por exemplo, você pode substituir a solução nas equações originais para verificar o resultado:

```
>> subs(equations, op(solution, 1))
```

$$\{2 = 2, 3 = 3\}$$

Você pode usar **assign (**op (solution, 1)) para nomear a solução de valores para os identificadores **x** e **z**.

No próximo exemplo, nós resolvemos duas equações polinomiais *não lineares* em várias incógnitas:

```
>> equations := {x^2 + y = 1, x - y = 2}:
>> solutions := solve(equations, {x, y})
```

$$\left\{\left[x = -\frac{\sqrt{13}}{2} - \frac{1}{2}, y = -\frac{\sqrt{13}}{2} - \frac{5}{2}\right], \left[x = \frac{\sqrt{13}}{2} - \frac{1}{2}, y = \frac{\sqrt{13}}{2} - \frac{5}{2}\right]\right\}$$

MuPAD encontra duas soluções distintas. Novamente, você pode usar **subs** para substituir as soluções em outras expressões:

```
>> map(subs(equations, op(solutions, 1)), expand),
   map(subs(equations, op(solutions, 2)), expand)
```

$$\{1 = 1, 2 = 2\}, \{1 = 1, 2 = 2\}$$

Frequentemente, soluções são representadas pelas expressões **RootOf**:

```
>> solve({x^3 + x^2 + 2*x = y, y^2 = x^3}, {x, y})
                              3
   {[x = 0, y = 0], [x = - y  + 4 y + 4,

                     4       3       2
      y in RootOf(X2  - 3 X2  - 2 X2  + 5 X2 + 8, X2)]}
```

Se você usar a expressão MaxDegree = n, então expressões **RootOf** para polinômios de grau acima para *n* são trocados por representações em termos de radicais se isto é possível. Notge que tais soluções tendem a ser bastante complicadas:

```
>> solve({x^3 + x^2 + 2*x = y, y^2 = x^3}, {x, y},
         MaxDegree = 4)
   {                    --     / /
   {                    |      | |
   { [x = 0, y = 0], |  x = | |
   {                    |      | \
   {                    |      |
   {                    |      |
   {                    |      |
   {                    --     \

                   1/2   1/2           1/3
      129 (5/18 I 3    73    + 3485/54)
      ---------------------------------- +
                     4

      ...
```

Especificando as incógnitas para resolver para isto é opcional. Para o caso especial onde você tem uma equação e uma incógnita, entretanto, o formato da saída depende em se você especifica a incógnita ou não. A regra geral é como segue:

A chamada da forma **solve (equation, unknown)**, onde **equação** é uma única equação (ou polinômio) e **unknown** é um identificador, retorna um objeto do MuPAD representando um conjunto de números. Todas as outras formas de **solve** para (uma ou várias) equações polinomiais retornam um conjunto de listas de equações ou uma expressão em envolvendo **RootOf.**

Aqui estão alguns exemplos:

```
>> solve(x^2 - 3*x + 2 = 0, x), solve(x^2 - 3*x + 2, x)
```
$\{1,2\}, \{1,2\}$
```
>> solve(x^2 - 3*x + 2 = 0), solve(x^2 - 3*x + 2)
```
$\{[x=1],[x=2]\}, \{[x=1],[x=2]\}$
```
>> solve({x^2 - 3*x + 2 = 0}, x),
   solve({x^2 - 3*x + 2}, x)
```
$\{[x=1],[x=2]\}, \{[x=1],[x=2]\}$
```
>> solve({x^2 - 3*x + y = 0, y - 2*x = 0}, {x, y})
```
$\{[x=0,y=0],[x=1,y=2]\}$
```
>> solve({x^2 - 3*x + y = 0, y - 2*x = 0})
```
$\{[x=0,y=0],[x=1,y=2]\}$

Se você não fornece incógnitas para resolver, **solve** usa internamente o sistema de função **indets** para encontrar os identificadores simbólicos nas equações e considera todas delas como incógnitas:

```
>> solve({x + y^2 = 1, x^2 - y = 0})
```
$\{[x = 1 - y^2, y \in \mathrm{RootOf}(\mathrm{X3}^4 - 2\,\mathrm{X3}^2 - \mathrm{X3} + 1, \mathrm{X3})]\}$

Por padrão, **solve** tenta encontrar todas as soluções *complexas* da(s) equação(ões). Se você apenas quer encontrar as soluções reais de uma única equação, use a opção Domain = Dom::Real ou assume (x, Type::Real):

```
>> solve(x^3 + x = 0, x)
```
$\{0, -i, i\}$
```
>> solve(x^3 + x = 0, x, Domain = Dom::Real)
```
$\{0\}$

Especificando Domain = Dom::Rational ou Domain = Dom::Integer, você obtêm apenas soluções racionais ou integrais, respectivamente. Se todos os números complexos satisfazem uma dada equação, então **solve** retorna $\mathbb{C}$, o conjunto de todos os números complexos:

```
>> solve(sin(x) = cos(x - PI/2), x)
```
$\mathbb{C}$
```
>> domtype(%)
```
solvelib::BasicSet

Existem quatro tais "conjuntos básicos" no MuPAD: as integrais $\mathbb{Z}$ (entrada como **Z_** na entrada do MuPAD), os números racionais $\mathbb{Q}$ (**Q_)**, os números reais $\mathbb{R}$ (**R_**), e os números complexos $\mathbb{C}$ (**C_**).

Você pode usar a função **float** para encontrar soluções *numéricas*. Contudo, com a alegação da forma

```
>> float(solve(equations, unknowns))
```

solve tenta primeiro resolver as equações simbolicamente. Se uma solução exata é encontrada, **float** converte o resultado para aproximações de pontos flutuantes. Se você quer computar de um modo puramente matemático, você pode usar **hold** (Seção 5.2) para esvaziar pré processamento simbólico:

```
>> float(hold(solve)({x^3 + x^2 + 2*x = y, y^2 = x^2},
                     {x, y}))
   {[x = - 0.5 - 1.658312395 I, y = 0.5 + 1.658312395 I],

      [x = - 0.5 + 1.658312395 I, y = 0.5 - 1.658312395 I

      ], [x = - 0.5 - 0.8660254038 I,

      y = - 0.5 - 0.8660254038 I],

      [x = - 0.5 + 0.8660254038 I,

      y = - 0.5 + 0.8660254038 I], [x = 0.0, y = 0.0]}
```

Alternativamente, a biblioteca **numeric** provê várias funções para resolver equações numéricas tais como **numeric::solve, numeric::fsolve,** ou **numeric::realroots**. O sistema de ajuda dá detalhes sobre essas rotinas: chame **?solvers** para um guia de sobrevivência ou veja **?numeric::solve** etc.

Exercício 8.1: Compute a solução geral do sistema de equações lineares

$$a + b + c + d + e = 1,$$
$$a + 2b + 3c + 4d + 5e = 2,$$
$$a - 2b - 3c - 4d - 5e = 2,$$
$$a - b - c - d - e = 3\ .$$

Quantos parâmetros livres a solução tem?

8.2 Equações Gerais e Inequações

O solucionador do MuPAD **solve** pode controlar uma série de equações (não polinomiais). Por exemplo, a equação $\exp(x) = 8$ tem infinitamente tantas soluções da forma $ln(8) + 2i\pi k$ com $k \in \mathbb{Z}$ no plano complexo:

```
>> S := solve(exp(x) = 8, x)
```
$$\{3\ln(2) + 2i\pi k \mid k \in \mathbb{Z}\}$$

O tipo de dado do resultado retornado é um também chamado “conjunto imagem.” Ele representa um conjunto matemático da forma $\{f(x) \mid x \in A\}$, onde A é outro conjunto:

```
>> domtype(S)
```
Dom::ImageSet

Esse tipo de dado é capaz de representar infinitamente vários elementos.

Se você omitir a variável para resolvê-la, o sistema retorna uma fórmula lógica, usando o operador do MuPAD **in**:

```
>> solve(exp(x) = 8)
```
$$x \in \{3\ln(2) + 2i\pi k \mid k \in \mathbb{Z}\}$$

Você pode usar **map** para aplicar uma função para um conjunto imagem:

```
>> map(S, _plus, -ln(8))
```
$$\{2i\pi k \mid k \in \mathbb{Z}\}$$

A função **is** (Seção 9.3) pode controlar conjuntos imagem:

```
>> S := solve(sin(PI*x/2) = 0, x)
```
$$\{2l \mid l \in \mathbb{Z}\}$$
```
>> is(1 in S), is(4 in S)
```
FALSE, TRUE

A equação **exp (x) = sin (x)** também tem infinitamente muitas soluções, as quais, entretanto, MuPAD não pode representar exatamente. Nesse caso, ele retorna a chamada para **solve** simbolicamente:

```
>> solutions := solve(exp(x) = sin(x), x)
```
$$\text{solve}\left(e^x - \sin(x) = 0, x\right)$$

Aviso: Em contraste com equações polinomiais, o solucionador numérico computa no máximo uma solução de uma equação não polinomial:

```
>> float(solutions)
```
$$\{-226.1946711\}$$

Contudo, você pode especificar uma gama de pesquisa para selecionar uma solução numérica particular:

```
>> numeric::solve(exp(x) = sin(x), x = -10..-9)
```
$$\{-9.424858654\}$$

Usando **numeric::realroots**, você encontra incluído para *todas* as raízes reais em um dado intervalo:

```
>> numeric::realroots(exp(x) = sin(x), x = -10..-5)
```
$$[[-9.43359375, -9.423828125], [-6.2890625, -6.279296875]]$$

MuPAD tem um tipo de dado especial para a solução de equações paramétricas: **piecewise**. Por exemplo, o conjunto de soluções $x \in \mathbb{C}$ da equação $(a\,x^2 - 4)(x - b) = 0$ leva em diferentes formas, dependendo do valor do parâmetro a:

```
>> delete a: p := solve((a*x^2 - 4)*(x - b), x)
```
$$\begin{cases} \{b\} & \text{if } a = 0 \\ \left\{b, -\frac{2}{\sqrt{a}}, \frac{2}{\sqrt{a}}\right\} & \text{if } a \neq 0 \end{cases}$$
```
>> domtype(p)
```
piecewise

A função **map** aplica uma função para todas as filiais de um objeto **piecewise:**

```
>> map(p, _power, 2)
```
$$\begin{cases} \{b^2\} & \text{if } a = 0 \\ \left\{\frac{4}{a}, b^2\right\} & \text{if } a \neq 0 \end{cases}$$

Depois da seguinte substituição, o objeto **piecewise** é simplificado a um conjunto:

```
>> eval(subs(%, [a = 4, b = 2]))
```
$$\{1, 4\}$$

A função **solve** também pode controlar inequações. Ela então retorna um intervalo ou uma união de intervalos, do tipo de domínio **Dom::Interval:**

```
>> solve(x^2 < 1, x)
```
$$(-1, 1)$$
```
>> domtype(%)
```
Dom::Interval
```
>> S := solve(x^2 >= 1, x)
```
$$[1, \infty) \cup (-\infty, -1]$$
```
>> is(-2 in S), is(0 in S)
```
TRUE, FALSE

8.3 Equações diferenciais

A função **ode** define uma equação diferencial ordinária. Um tal objeto tem dois componentes: uma equação e a função para resolver.

```
>> diffEquation := ode(y'(x) = y(x)^2, y(x))
```
$$\text{ode}\left(\frac{\partial}{\partial x} y(x) - y(x)^2, y(x)\right)$$

A seguinte chamada para **solve** encontra a solução geral contendo uma constante arbitrária C_3:

```
>> solve(diffEquation)
```
$$\left\{0, \frac{1}{C3 - x}\right\}$$

Equações diferenciais de alta ordem podem ser manipulados também:

```
>> solve(ode(y''(x) = y(x), y(x)))
```
$$\left\{C5\, e^{-x} + C6\, e^{x}\right\}$$

Você pode especificar condições iniciais ou valores limite passando a equação diferencial junto com as condições inicial/limite como um conjunto quando chamada **ode**:

```
>> diffEquation :=
   ode({y''(x) = y(x), y(0) = 1, y'(0) = 0}, y(x)):
```

MuPAD agora ajusta asa constantes livres na solução geral de acordo com as condições iniciais/limites:

```
>> solve(diffEquation)
```
$$\left\{\frac{e^{x}}{2} + \frac{e^{-x}}{2}\right\}$$

Você pode especificar sistemas de equações com várias funções na forma de um conjunto:

```
>> solve(ode({y'(x) = y(x) + 2*z(x), z'(x) = y(x)},
             {y(x), z(x)}))
```
$$\left\{\left[z(x) = \frac{C11\, e^{2x}}{2} - C10\, e^{-x}, y(x) = C10\, e^{-x} + C11\, e^{2x}\right]\right\}$$

A função **numeric::odesolve** da biblioteca **numeric** resolve a equação diferencial ordinária $Y'(x) = f(x, Y(x))$ com a condição inicial $Y(x_o) = Y_0$ numericamente. Você pode fornecer o lado da mão direita da equação diferencial como uma função $f(x, Y)$ de dois argumentos, onde x é uma escalar e Y é um vetor. Se você combinar os componentes y e z no exemplo anterior para um vetor $Y = (y, z)$, você pode definir o lado da mão direita da equação

$$\frac{d}{dx} Y = \frac{d}{dx} \begin{pmatrix} y \\ z \end{pmatrix} = \begin{pmatrix} y + 2z \\ y \end{pmatrix} = \begin{pmatrix} \texttt{Y[1]} + 2 \cdot \texttt{Y[2]} \\ \texttt{Y[1]} \end{pmatrix} =: \texttt{f(x, Y)}$$

Na forma

```
>> f := (x, Y) -> [Y[1] + 2*Y[2], Y[1]]:
```

Note que $f(x, Y)$ deve ser um vetor. Aqui, isso é realizado pelos sentidos de uma lista contendo os componentes no lado direito da equação diferencial. A chamada

```
>> numeric::odesolve(0..1, f, [1, 1])
```
[9.729448318, 5.04866388]

Integra o sistema de equações diferenciais com os valores iniciais $Y(0) = (y(0), z(0)) = (1,1)$ sobre o intervalo $x \in [0,1]$; as condições iniciais são especificadas pela lista [1, 1]. O solucionador numérico retorna a solução numérica vetor $Y(1) = (y(1), z(1))$como uma lista.

Exercício 8.2: Verifique as soluções numéricas $y(1) = 9.729\ldots$ e $z(1) = 5.048\ldots$ do sistema de equações diferenciais

$$y'(x) = y(x) + 2\, z(x),\ z'(x) = y(x)$$

Computados acima pela substituição dos valores iniciais $y(0) = 1$, $z(0) = 1$ na solução geral simbólica, determinando os valores para as constantes livres, e avaliando a solução simbólica em $x = 1$.

Exercício 8.3:

1) Compute a solução geral $y(x)$ da equação diferencial $y\text{^}' = y\text{^}2/x$.
2) Determine a solução $y(x)$ para cada dos seguintes problemas de valor inicial:

a) $y' - y\sin(x) = 0, y'(1) = 1,$

b) $2y' + \frac{y}{x} = 0, y'(1) = \pi.$

3) Encontre a solução geral do seguinte sistema de equações diferenciais ordinárias em $x(t)$, $y(t)$, $z(t)$:

$x' = -3\,y\,z,\ y' = 3\,x\,z,\ z' = -x\,y.$

8.4 Equações recorrentes

Equações recorrentes são equações para funções dependendo em um parâmetro discreto (um "índice"). Você pode gerar tal um objeto com a função **rec,** a qual argumentos são uma equação, a função para ser determinada e, opcionalmente, um conjunto de condições iniciais:

```
>> equation := rec(y(n + 2) = y(n + 1) + 2*y(n), y(n)):
>> solve(equation)
```

$$\{(-1)^n\ C1 + 2^n\ C2\}$$

A solução geral contém duas constantes arbitrárias (C_1, C_2 neste caso), do qual é satisfatório ajustado quando você especifica as condições iniciais:

```
>> solve(rec(y(n + 2) = 2*y(n) + y(n + 1), y(n),
             {y(0) = 1}))
```

$$\{2^n\ C4 - (-1)^n\ (C4 - 1)\}$$

```
>> solve(rec(y(n + 2) = 2*y(n) + y(n + 1), y(n),
             {y(0) = 1, y(1) = 1}))
```

$$\left\{\frac{(-1)^n}{3} + \frac{2\,2^n}{3}\right\}$$

Exercício 8.4: os números de Fibonacci são definidos pela recorrência $F_n = F_{n-1} + F_{n-2}$ com os valores iniciais $F_0 = 0$, $F_{1 = 1.}$ Use **solve** para encontrar uma representação explícita para F_n.

Capítulo 9 Manipulando expressões

Quando avalia objetos, MuPAD automaticamente executa uma variedade de simplificações. Por exemplo, operações aritméticas entre integrais são executadas ou exp (ln(x)) é simplificado para *x*. Outras simplificações tais como sin (*x*)^2 + cos (*x*)^2 = 1, ln (exp (*x*)) = *x*, (*x*^2 − 1)/*x* − 1) = *x* + 1, ou $\sqrt{x^2} = x$ n ao acontecem automaticamente. A razão é para um que muitas dessas rfegras não são universalmente válidas: por exemplo, $\sqrt{x^2} = x$ está errado para *x* = −2. Outras simplificações tais como sin(*x*)2 + cos(*x*)2 = 1 são universalmente válidas, mas haveria uma perda significativa de eficiência se MuPAD for sempre esquadrinhar expressões para a ocorrência de termos de sin e cos.

Além disso, não está claro em geral quais de várias representações matemáticas equivalentes é a mais apropriada. Por exemplo, isso deve ser razoável para substituir uma expressão tal como sin (*x*) por sua representação exponencial complexa

$$\sin(x) = -\frac{\mathrm{i}}{2}\,\exp(x\,\mathrm{i}) + \frac{\mathrm{i}}{2}\,\exp(-x\,\mathrm{i}).$$

Em uma tal situação, você pode controlar a manipulação e simplificação de expressões aplicando explicitamente sistema de funções apropriados. MuPAD provê as seguintes funções, as quais nós temos discutido em parte na Seção 2.3:

```
collect   :  collecting coefficients
combine   :  combining subexpressions
expand    :  expansion
factor    :  factorization
normal    :  normalization of rational expressions
partfrac  :  partial fraction decomposition
radsimp   :  simplification of radicals
rectform  :  Cartesian representation of complex values
rewrite   :  applying mathematical identities
simplify  :  universal simplifier
```

9.1 Transformando expressões

Se você entrar com o comando **collect(expressiom, unknown)**, o sistema considera a expressão como um polinômio na incógnita específica e agrupa os coeficientes de iguais potências:

```
>> x^2 + a*x + sqrt(2)*x + b*x^2 + sin(x) + a*sin(x):
>> collect(%, x)
```

$$(b+1)\,x^2 + \left(a+\sqrt{2}\right)\,x + (\sin(x) + a\,\sin(x))$$

Você pode especificar várias "incógnitas," as quais podem por si mesmas serem expressões, como uma lista:

```
>> collect(%, [x, sin(x)])
```

$$(b+1)\,x^2 + \left(a+\sqrt{2}\right)\,x + (a+1)\,\sin(x)$$

A função **combine(expressão, opção)** combina subexpressões usando identidades matemáticas entre funções indicadas por **option.** As opções implementadas são **arctan, exp, ln, sincos, sinhcosh.**

Por padrão, **combine** apenas emprega as identidades $a^b a^c = a^{b+c}$, $a^c b^c\ (ab)^c$, e $(a^b)^c = a^{bc}$ para potências, onde elas são válidas:
O inverso arctan da função tangente satisfaz a seguinte identidade:

```
>> f := x^(n + 1)*x^PI/x^2: f = combine(f)
```

$$\frac{x^{n+1}\,x^{\pi}}{x^2} = x^{\pi+n-1}$$

```
>> f := a^x*3^y/2^x/9^y: f = combine(f)
```

$$\frac{3^y\,a^x}{2^x\,9^y} = \left(\frac{1}{3}\right)^y\,\frac{a^x}{2}$$

```
>> combine(sqrt(6)*sqrt(7)*sqrt(x))
```

$$\sqrt{42\,x}$$

```
>> f := (PI^(1/2))^x: f = combine(f)
```

$$\sqrt{\pi}^x = \pi^{\frac{x}{2}}$$

```
>> f := arctan(x) + arctan(y): f = combine(f, arctan)
```

$$\arctan(x) + \arctan(y) = -\arctan\left(\frac{x+y}{x\,y-1}\right)$$

Para a função exponencial, nós temos exp (*x*) exp (*y*) = exp (*x* + *y*):

```
>> combine(exp(x)*exp(y)^2/exp(-z), exp)
```

$$e^{x+2y+z}$$

Note, contudo, que a identidade exp $(x)^y$ = exp (*x y*) conhecido para segurar para reais valores de *x* não segura ao longo do do plano complexo. Consequentemente, sem suposições em *x*, MuPAD não combina tais termos:

```
>> combine(exp(x)^y, exp)
```

$$(e^x)^y$$

Com certas suposições em *x* e *y*, o logaritmo satisfaz as regras ln (x) + ln (y) = ln (x y) e x ln (y) = ln (y^x):

```
>> combine(ln(x) + ln(2) + 3*ln(3/2), ln)
```

$$\ln\left(\frac{27\,x}{4}\right)$$

As funções trigonométricas satisfazem uma variedade de identidades que o sistema emprega para combinar produtos:

```
>> combine(sin(x)*cos(y), sincos),
   combine(sin(x)^2, sincos)
```

$$\frac{\sin(x-y)}{2} + \frac{\sin(x+y)}{2},\ \frac{1}{2} - \frac{\cos(2\,x)}{2}$$

Regras similares são aplicadas para as funções hiperbólicas:

```
>> combine(sinh(x)*cosh(y), sinhcosh),
   combine(sinh(x)^2, sinhcosh)
```

$$\frac{\sinh(x+y)}{2} + \frac{\sinh(x-y)}{2},\ \frac{\cosh(2\,x)}{2} - \frac{1}{2}$$

A função **expand** aplica as identidades usadas por **combine** na direção contrária: ela transforma chamadas de funções especiais com argumentos compostos para somas ou produtos de chamadas de função com argumentos mais simples via “adição de teoremas:”

```
>> expand(x^(y + z)), expand(exp(x + y - z + 4)),
   expand(ln(2*PI*x*y))
```

$$x^y\,x^z,\ \frac{e^4\,e^x\,e^y}{e^z},\ \ln(2) + \ln(\pi) + \ln(x\,y)$$

```
>> expand(sin(x + y)), expand(cosh(x + y))
```

$$\cos(x)\,\sin(y) + \cos(y)\,\sin(x),\ \cosh(x)\,\cosh(y) + \sinh(x)\,\sinh(y)$$

```
>> expand(sqrt(42*x*y))
```

$$\sqrt{42}\,\sqrt{x\,y}$$

Aqui, o sistema não executa certas “expansões” tais como ln (x y) = ln (x) +ln (y), desde que tais uma identidade controla apenas sobre suposições adicionais (por exemplo, para real positivo *x* e *y*).

O uso mais frequente de **expand** trabalha recursivamente para todas as subexpressões:

```
>> expand((x - y)*(x + y)*sin(exp(x + y + z)))
```

$$x^2\,\sin(e^x\,e^y\,e^z) - y^2\,\sin(e^x\,e^y\,e^z)$$

Você pode fornecer expressões como argumentos adicionais para **expand**. Estas subexpressões *não* são expandidas:

```
>> expand((x - y)*(x + y)*sin(exp(x + y + z)),
          x - y, x + y + z)
```

$$x \sin\left(e^{x+y+z}\right)\,(x-y) + y \sin\left(e^{x+y+z}\right)\,(x-y)$$

A função **factor** fatora polinômios e expressões:

```
>> factor(x^3 + 3*x^2 + 3*x + 1)
```

$$(x+1)^3$$

Aqui o sistema fatora “acima dos números racionais:” isso parece para fatores polinomiais com coeficientes numéricos racionais. Com efeito, MuPAD não retorna a fatoração $x2 - 2 = (x - \sqrt{2})(\overline{x} + \sqrt{2})$: ¯

```
>> factor(x^2 - 2)
```

$$\left(x^2 - 2\right)$$

Para somas de expressões racionais, **factor** primeiro computa um denominador comum e então fatora ambos o numerador e o denominador:

```
>> f := (x^3 + 3*y^2)/(x^2 - y^2) + 3: f = factor(f)
```

$$\frac{x^3 + 3\,y^2}{x^2 - y^2} + 3 = \frac{x^2 \cdot (x+3)}{(x-y)\cdot(x+y)}$$

MuPAD pode fatorar não apenas polinômios e funções racionais. Para expressões mais gerais, o sistema internamente substitui subexpressões tais como chamadas de função simbólica por identificadores, fatora o polinômio correspondente ou função racional, e resubstitui os identificadores temporários:

```
>> factor((exp(x)^2 - 1)/(cos(x)^2 - sin(x)^2))
```

$$\frac{(e^x - 1)\cdot(e^x + 1)}{(\cos(x) - \sin(x))\cdot(\cos(x) + \sin(x))}$$

Não obstante, **normal** cancela fatores comuns no numerador e denominador:

```
>> f := x^2/(x + y) - y^2/(x + y): f = normal(f)
```

$$\frac{x^2}{x+y} - \frac{y^2}{x+y} = x - y$$

Como **factor**, **normal** pode controlar expressões arbitrárias:

```
>> f := (exp(x)^2-exp(y)^2)/(exp(x)^3 - exp(y)^3):
>> f = normal(f)
```

$$\frac{(e^x)^2 - (e^y)^2}{(e^x)^3 - (e^y)^3} = \frac{e^x + e^y}{(e^x)^2 + e^x\,e^y + (e^y)^2}$$

A função partfrac decompõe uma expressão racional dentro de uma parte polinomial mais um a soma de termos racionais nos quais o grau do numerador é menor que o grau do denominador correspondente (decomposição parcial de fração):

```
>> f := x^2/(x^2 - 1): f = partfrac(f, x)
```

$$\frac{x^2}{x^2-1} = \frac{1}{2\,(x-1)} - \frac{1}{2\,(x+1)} + 1$$

Os denominadores dos termos são os fatores que MuPAD encontra quando fatora o denominador comum:

```
>> denominator := x^5 + x^4 - 7*x^3 - 11*x^2 - 8*x - 12:
>> factor(denominator)
```

$$(x-3)\cdot\left(x^2+1\right)\cdot(x+2)^2$$

```
>> partfrac(1/denominator, x)
```

$$\frac{\frac{9\,x}{250} - \frac{13}{250}}{x^2+1} - \frac{1}{25\,(x+2)} - \frac{1}{25\,(x+2)^2} + \frac{1}{250\,(x-3)}$$

Outra função para manipulação de expressões é **rewrite**. Ela emprega identidades para eliminar certas funções completamente de uma expressão e substitui elas por outras funções. Por exemplo, você sempre pode expressar sin e cos por tan com o argumento metade:

$$\sin(x) = \frac{2\,\tan(x/2)}{1+\tan(x/2)^2}, \quad \cos(x) = \frac{1-\tan(x/2)^2}{1+\tan(x/2)^2}.$$

As funções trigonométricas também são relatadas à função exponencial complexa:

$$\sin(x) = -\frac{\mathrm{i}}{2}\,\exp(\mathrm{i}\,x) + \frac{\mathrm{i}}{2}\,\exp(-\mathrm{i}\,x)\ ,$$

$$\cos(x) = \frac{1}{2}\,\exp(\mathrm{i}\,x) + \frac{1}{2}\,\exp(-\mathrm{i}\,x)\ .$$

Você pode expressar as funções hiperbólicas e suas funções inversas nos termos da função exponencial e o logaritmo:

$$\sinh(x) = \frac{\exp(x) - \exp(-x)}{2}\ , \quad \cosh(x) = \frac{\exp(x) + \exp(-x)}{2}\ ,$$

$$\operatorname{arcsinh}(x) = \ln(x + \sqrt{x^2+1})\ , \quad \operatorname{arccosh}(x) = \ln(x + \sqrt{x^2-1})\ .$$

A chamada da forma **rewrite** (**expressão, opção)** emprega essas identidades. Tabela 9.1 lista as regras implementadas no MuPAD.

target : *function(s)* → *rewritten*
in terms of

andor	: logical operators xor,	→ and, or,

arccos arccosh arccot arccoth arcsin arcsinh arctan arctanh	==>, <=> : inverse trig. functions : inverse trig. functions, ln : inverse trig. functions : inverse trig. functions, ln : inverse trig. functions : inverse trig. functions, ln : inverse trig. functions : inverse trig. functions, ln	not → arccos → arccosh → arccot → arccoth → arcsin → arcsinh → arctan → arctanh → bernoulli → cos
bernoulli : euler		
cos	: exponential function exp, trig. and hyperbolic functions	

cosh : exponential function exp, trig. and hyperbolic functions
→ cosh
cot : exponential function exp, trig. and hyperbolic functions
→ cot
coth : exponential function exp, trig. and hyperbolic functions
→ coth
diff : differential operator D → diff
D : differentiating function diff → D
exp : powers (^), trig. and hyperbolic functions and their inverses, polar angle arg, dawson
→ exp, ln
erf : dawson → erf
fact : Γ-function gamma, double factorial fact2, binomial coefficients binomial, *β*-function beta, pochhammer
→ fact
gamma : factorial fact, double factorial fact2, binomial coefficients binomial, *β*-function beta, pochhammer
→ gamma

Table 9.1a: Targets of rewrite

9-

```
>> rewrite(D(D(u))(x), diff)
```

$$\frac{\partial^2}{\partial x^2} u\,(x)$$

```
>> rewrite(sin(x)/cos(x), exp) = rewrite(tan(x), exp)
```

$$\frac{\frac{\mathrm{i}}{2}\,\mathrm{e}^{-\mathrm{i}\,x} - \frac{\mathrm{i}}{2}\,\mathrm{e}^{\mathrm{i}\,x}}{\frac{\mathrm{e}^{-\mathrm{i}\,x}}{2} + \frac{\mathrm{e}^{\mathrm{i}\,x}}{2}} = -\frac{\mathrm{i}\,\left(\mathrm{e}^{\mathrm{i}\,x}\right)^2 - \mathrm{i}}{\left(\mathrm{e}^{\mathrm{i}\,x}\right)^2 + 1}$$

```
>> rewrite(arcsinh(x) - arccosh(x), ln)
```

$$\ln\left(x + \sqrt{x^2+1}\right) - \ln\left(x + \sqrt{x^2-1}\right)$$

Para expressões representando *números* complexos, você pode facilmente computar as partes real e imaginária usando **Re** e **Im** são retornadas simbolicamente:

```
>> Re(a*b + I), Im(a*b + I)
```

$$\Re\,(a\,b)\,,\ \Im\,(a\,b) + 1$$

Em um tal caso, você pode usar a função **rectform** (abreviação para: forma retangular) para decompor a expressão dentro de parte real e imaginária. O nome desta função é derivado do fato que ela computa as coordenadas do retangular usual (Cartesiano) sistema de coordenadas. MuPAD decompõe os símbolos contidos na expressão dentro de suas partes reais e imaginárias e expressa o resultado final de acordo:

```
>> rectform(a*b + I)
```

$$\Re\,(a)\ \Re\,(b) - \Im\,(a)\ \Im\,(b) + \mathrm{i}\,\Im\,(a)\ \Re\,(b) + \mathrm{i}\,\Im\,(b)\ \Re\,(a) + \mathrm{i}$$

```
>> rectform(exp(x))
```

$$\cos\,(\Im\,(x))\ \mathrm{e}^{\Re(x)} + \mathrm{i}\,\mathrm{e}^{\Re(x)}\ \sin\,(\Im\,(x))$$

Novamente, você pode extrair as partes reais e imaginárias do resultado com **Re** e **Im**, respectivamente:

```
>> Re(%), Im(%)
```

$$\cos\,(\Im\,(x))\ \mathrm{e}^{\Re(x)},\ \mathrm{e}^{\Re(x)}\ \sin\,(\Im\,(x))$$

Como um princípio básico, **rectform** considera todos os identificadores simbólicos como representantes de números complexos. Contudo, você pode usar **assume** (Seção 9.3) para especificar que um identificador representa apenas números reais:

```
>> assume(a, Type::Real):
   z := rectform(a*b + I)
```
$$a\,\Re(b) + \mathrm{i}\,a\,\Im(b) + \mathrm{i}$$

Os resultados de **rectform** tem um tipo de dado especial:

```
>> domtype(z)
```
rectform

Você pode usar a função **expr** para converter um objeto para uma expressão “normal” do MuPAD do tipo de domínio **DOM_EXPR:**

```
>> expr(z)
```
$$\mathrm{i}\,a\,\Im(b) + a\,\Re(b) + \mathrm{i}$$

Nós recomendamos que você aplique a função **rectform** apenas para expressões contendop identificadores simbólicos. Para expressões sem tais identificadores, **Re** e **Im** retornam a decomposição dentro da parte real e imaginária muito rapidamente.

9.2 Simplificando Expressões

Em alguns casos, uma transformação leva a uma expressão mais simples:

```
>> f := 2^x*3^x/8^x/9^x: f = combine(f)
```
$$\frac{2^x\,3^x}{8^x\,9^x} = \left(\frac{1}{12}\right)^x$$
```
>> f := x/(x + y) + y/(x + y): f = normal(f)
```
$$\frac{x}{x+y} + \frac{y}{x+y} = 1$$

Para este fim, contudo, você deve inspecionar a expressão e decidir por si mesmo qual função usar para simplificação. Há ferramentas para aplicação de várias simplificações e algoritmos para uma expressão *automaticamente:* as funções **simplify** e **Simplify**. Esses são simplificadores universais os quais MuPAD usa para alcançar uma representação de uma expressão que é tão “simples” quanto possível:

```
>> f := 2^x*3^x/8^x/9^x: f = simplify(f)
```
$$\frac{2^x\,3^x}{8^x\,9^x} = \left(\frac{1}{12}\right)^x$$
```
>> f := (1 + (sin(x)^2 + cos(x)^2)^2)/sin(x):
>> f = simplify(f)
```
$$\frac{\left(\cos(x)^2 + \sin(x)^2\right)^2 + 1}{\sin(x)} = \frac{2}{\sin(x)}$$
```
>> f := x/(x + y) + y/(x + y) - sin(x)^2 - cos(x)^2:
>> f = simplify(f)
```
$$\frac{x}{x+y} + \frac{y}{x+y} - \cos(x)^2 - \sin(x)^2 = 0$$
```
>> f := (exp(x) - 1)/(exp(x/2) + 1): f = simplify(f)
```
$$\frac{\mathrm{e}^x - 1}{\mathrm{e}^{\frac{x}{2}} + 1} = \mathrm{e}^{\frac{x}{2}} - 1$$
```
>> f := sqrt(997) - (997^3)^(1/6): f = simplify(f)
```
$$\sqrt{997} - \sqrt[6]{991026973} = 0$$

Note que **simplify** opera em um modo puramente heurístico desde não haja resposta geral ao que “simples” signifique. Você pode controlar o processo de simplificação fornecendo argumentos adicionais. Como no caso de **combine**, você pode requerer simplificações particulares pelos sentidos das opções. Por exemplo, você pode dizer ao simplificador explicitamente para simplificar expressões contendo raízes quadradas:

```
>> f := sqrt(4 + 2*sqrt(3)):
   f = simplify(f, sqrt)
```
$$\sqrt{\sqrt{3} + 2\sqrt{2}} = \sqrt{3} + 1$$

As opções possíveis são **exp, ln, cos, sin, sqrt, logic,** e **relation**. Internamente, **simplify** então confina a si mesmo para essas regras de simplificação que são válidas para a função dada como opção. A opção **logic** e **relation** são para simplificar expressões lógicas e equações e inequações, respectivamente (veja também a página de ajuda correspondente: **?simplfy**).

Ao invés de **simplify (expression, sqrt)**, você pode usar a função **radsimp** para simplificar expressões numéricas contendo raízes quadradas ou outros radicais:

```
>> f = radsimp(f)
```
$$\sqrt{\sqrt{3} + 2\sqrt{2}} = \sqrt{3} + 1$$
```
>> f := 2^(1/4)*2 + 2^(3/4) - sqrt(8 + 6*2^(1/2)):
>> f = radsimp(f)
```
$$2\sqrt[4]{2} - \sqrt{3\sqrt{2} + 4\sqrt{2} + \sqrt[4]{8}} = 0$$

Em muitos casos, usando **simplify** sem opções é apropriado. Contudo, uma tal chamada frequentemente consome muito tempo, desde que o algoritmo de simplificação é bastante complexo. Isso pode ser útil para especificar opções adicionais para salvar tempo computando, desde que então simplificações são executadas apenas para funções especiais.

O segundo simplificador geral, **Simplify** (note o S maiúsculo), é frequentemente mais lento que **simplify**, mas muito mais poderoso e permite sintonização muito melhor:

```
>> f := (tanh(x/2) + 1)/(1 - tanh(x/2)):
   f, simplify(f), Simplify(f), Simplify(f, Steps = 100)
```

$$-\frac{\tanh\left(\frac{x}{2}\right)+1}{\tanh\left(\frac{x}{2}\right)-1},\ -\frac{\tanh\left(\frac{x}{2}\right)+1}{\tanh\left(\frac{x}{2}\right)-1},\ \left(e^{\frac{x}{2}}\right)^2,\ e^x$$

Aqui, nós usamos a opção **Steps** para avisar a **Simplify** quantos passos "primários" devem ser tentados antes de se render a pesquisa para uma expressão mais simples. No fim, as maiorias de todos os passos levam a lugar nenhum; nós podemos perguntar **Simplify** para uma lista de passos executados e ver que apenas três passos foram usados atualmente:

```
>> Simplify(f, Steps = 100, OutputType = "Proof")
   Input was -1/(tanh(1/2*x) - 1)*(tanh(1/2*x) + 1).
   Applying the rule X -> rewrite(X, exp)(X) to -1/(tanh\
   (1/2*x) - 1)*(tanh(1/2*x) + 1) gives -1/((exp(1/2*x)^\
   2 - 1)/(exp(1/2*x)^2 + 1) - 1)*((exp(1/2*x)^2 - 1)/(e\
   xp(1/2*x)^2 + 1) + 1)
   Applying the rule X -> normal(X)(X) to -1/((exp(1/2*x\
   )^2 - 1)/(exp(1/2*x)^2 + 1) - 1)*((exp(1/2*x)^2 - 1)/\
   (exp(1/2*x)^2 + 1) + 1) gives exp(1/2*x)^2
   Applying the rule proc combine::exp(e) ... end(X) to \
   exp(1/2*x)^2 gives exp(x)
   END OF PROOF
```

Para isto (admitidamente difícil de ler) a saída sugere, **Simplify** trabalha por seguir uma *regra base*. A documentação, acessível em **?Simplify:details** (note a única coluna!), mostra exemplos com conjuntos de aplicação específica.

Mais futuramente, o usuário pode controlar a ideia de **Simplify** da qual é "simples." O exercício 9.4 mostra uma aplicação.

Exercício 9.1: É possível reescrever produtos de funções trigonométricas como combinações lineares de termos de seno e coseno quais argumentos são múltiplas integrais dos argumentos originais (expansão de Fourier). Encontre constantes *a*, *b*, *c*, *d*, e
e tais que

$\cos(x)2 + \sin(x)\cos(x) = a + b\sin(x) + c\cos(x) + d\sin(2x) + e\cos(2x)$ mantém

Exercício 9.2: Use MuPAD para provar as seguintes identidades:

1) $$\frac{\cos(5\,x)}{\sin(2\,x)\cos^2(x)} = -5\sin(x) + \frac{\cos^2(x)}{2\sin(x)} + \frac{5\sin^3(x)}{2\cos^2(x)}\ ,$$

2) $$\frac{\sin^2(x) - e^{2\,x}}{\sin^2(x) + 2\sin(x)\,e^x + e^{2\,x}} = \frac{\sin(x) - e^x}{\sin(x) + e^x}\ ,$$

3) $$\frac{\sin(2\,x) - 5\sin(x)\cos(x)}{\sin(x)\,(1+\tan^2(x))} = -\frac{9\cos(x)}{4} - \frac{3\cos(3\,x)}{4}\ ,$$

4) $$\sqrt{14 + 3\sqrt{3 + 2\sqrt{5 - 12\sqrt{3 - 2\sqrt{2}}}}} = \sqrt{2} + 3\ .$$

Exercício 9.3: MuPAD computa as seguintes integrais de **f**:

```
>> f := sqrt(sin(x) + 1): int(%, x)
```

$$\frac{2\,(\sin(x)-1)\,\sqrt{\sin(x)+1}}{\cos(x)}$$

Essa derivada não é literalmente idêntica ao integrando:

```
>> diff(%, x)
```

$$\frac{\sin(x)-1}{\sqrt{\sin(x)+1}} + 2\sqrt{\sin(x)+1} + \frac{2\sin(x)\,(\sin(x)-1)\,\sqrt{\sin(x)+1}}{\cos(x)^2}$$

Simplifique essa expressão.

Exercício 9.4: Usando a opção **Valuation**, nós podemos passar uma função para **Simplify**para determinar quais expressões são "simples:" a função é chamada com uma expressão como seu argumento e retorna um número, números mais altos para expressõers mais complexas.

Deixe-nos tentar escrever $\tan(x) - \cos(x)$ sem a função tangente. O primeiro, algo ingênuo aproxima é para usar uma avaliação de função que simplesmente parece para a presença de **tan**.

```
>> noTangent := x -> if has(x, hold(tan))
                     then 1
                     else 0 end_if:
   Simplify(tan(x) - cot(x), Valuation = noTangent)
                                    2
                  2 I (exp(1/2 I x)  - 1)
   - ------------------------------------------------ - ...
     /              2     2     \
     | (exp(1/2 I x)  - 1)      |              2
     | -------------------- + 1 | (exp(1/2 I x)  + 1)
     |              2     2     |
     \ (exp(1/2 I x)  + 1)      /
```

Agora, esta expressão atualmente não contém **tan**, mas é dificilmente "simples." Melhore **noTangent** tal que a chamada acima retorna uma expressão simples sem **tan** e **cot**. Você deve querer informar a si mesmo sobre **lenght** primeiro ou usar **Simplify::defaultValuation** ao invés disso, a qual é a função usada por **Simplify** se nada for especificado.

9.3 Suposições sobre identificadores simbólicos

MuPAD executa transformações ou simplificações para objetos contendo identificadores simbólicos apenas se as regras correspondentes aplicam-se no plano inteiro complexo. Contudo, algumas regras familiares para computar com números reais não são válidas geralmente para números complexos. Por exemplo, a raiz quadrada e o logaritmo são funções complexas multi avaliadas, e as funções do MuPAD internamente fazem certas suposições sobre os cortes de filial.

$\ln(e^x) = x$	for real x
$\ln(x^n) = n\,\ln(x)$	for real $x > 0$
$\ln(x\,y) = \ln(x) + \ln(y)$	for real $x > 0$ or real $y > 0$
$\sqrt{x^2} = \lvert x\rvert$	for real x
$e^{x/2} = (e^x)^{1/2}$	for real x

Você pode usar a função **assume** para dizer ao sistema funções tais como **expand, simplify, limit, solve**, e **int** que elas fazem certas suposições sobre o sentido de certos identificadores. Nós apenas demonstramos alguns exemplos simples aqui. Você encontra mais informação na página de ajuda correspondente: **?property**.

Você pode usar um tipo especificador (Capítulo 15) para dizer ao MuPAD que um identificador simbólico representa apenas valores correspondentes ao sentido matemático do tipo. Por exemplo, os comandos

```
>> assume(x, Type::Real): assume(y, Type::Real):
   assume(n, Type::Integer):
```

restringem *x* e *y* para serem números reais e *n* para ser uma integral. Agora **simplify** pode aplicar regras adicionais:

```
>> simplify(ln(exp(x))), simplify(sqrt(x^2))
    x, x sign (x)
```

O comando **assume (x, Type::Positive)** ou, equivalentemente:

```
>> assume(x > 0):
```

Restringe *x* para números positivos reais. Depois da suposição, um obtém:

```
>> simplify(ln(x^n)), simplify(ln(x*y) - ln(x) - ln(y)),
   simplify(sqrt(x^2))
    n ln (x) , 0, x
```

Transformações e simplificações com constantes são executadas sem suposições adicionais desde que seu sentido matemático é conhecido:

```
>> expand(ln(2*PI*z)), sqrt((2*PI*z)^2)
    ln (2) + ln (π) + ln (z) , 2 π √(z^2)
```

Os operadores aritméticos levam dentro da conta certas propriedades matemáticas automaticamente:

```
>> (a*b)^m
    (a b)^m
>> assume(m, Type::Integer): (a*b)^m
    a^m b^m
```

A função **is** checa se um objeto do MuPAD tem uma certa propriedade matemática:

```
>> is(1, Type::Integer), is(PI + 1, Type::Real)
    TRUE, TRUE
```

Em adição, **is** leva em conta as propriedades matemáticas dos identificadores conjuntos por **assume:**

```
>> delete x: is(x, Type::Integer)
    UNKNOWN
>> assume(x, Type::Integer):
>> is(x, Type::Integer), is(x, Type::Real)
    TRUE, TRUE
```

Aqui, o valor Booleano **UNKNOWN** expressa que o fato que o sistema não pode decidir se **x** representa uma integral ou não.

Em contraste com **is**, a função **testtype** apresentada na Seção 15.1 checa o tipo *técnico* de um objeto do MuPAD:

```
>> testtype(x, Type::Integer), testtype(x, DOM_IDENT)
    FALSE, TRUE
```

Questões da seguinte forma também são possíveis:

```
>> assume(y > 5): is(y + 1 > 4)
    TRUE
```

A função **getprop** retorna uma propriedade matemática de um identificador ou uma expressão:

```
>> getprop(y), getprop(y^2 + 1)
    (5, ∞) , (26, ∞)
```

Você pode deletar as propriedades de um identificador usando **unassume** da palavra chave **delete**:

```
>> delete y: is(y > 5)
    UNKNOWN
```

Se nenhum dos identificadores em uma expressão tem qualquer propriedade fixada à ela, então **getprop** retorna a expressão por si mesma:

```
>> getprop(y), getprop(y^2 + 1)
    y, y^2 + 1
>> getprop(3), getprop(sqrt(2) + 1)
    3, √2 + 1
```

Uma exceção dessa regra acontece quando a expressão contém uma das funções **Re, Im,** ou **abs**(x) é sempre não negativo, até mesmo se a expressão **ex** não envolve quaisquer identificadores com propriedades:

```
>> getprop(Re(y)), getprop(abs(y^2 + 1))
    ℝ, [0, ∞)
>> is(abs(y^2 + 1), Type::Real)
    TRUE
```

Há quatro tipos de propriedades matemáticas disponíveis no muPAD

- domínios de número básico, tais como integrais, os números racionais, os números reais, os números positivos reais, ou os números primos,
- intervalos de domínios de número básico,
- classe resídua de integrais, e
- relação entre um identificador e uma expressão

arbitrária. Essas classes são sumarizadas na tabela 9.2.

Se T é um tipo especificado para um domínio básico numérico, um intervalo, ou uma classe residual da coluna do meio da Tabela 9.2, então **assume**(x,T) fixa a propriedade matemática "é um elemento de S" para o identificador **x**, onde *S* é o conjunto correspondente na coluna da direita. Similarmente, **is(ex, T)** checa se a expressão **ex** pertence ao conjunto *S* matematicamente. A sintaxe para relações é mais intuitiva: por exemplo, **assume (x < b)** fixa a propriedade matemática "é menor que *b* "para o identificador **x**, e **is (a < b)** checa se a relação $a < b$ mantém verdadeiro matematicamente para a expressão **a** e **b**.

Há frequentemente vários modos equivalentes para especificar uma propriedade. Por exemplo, **Type::Positive, Type::Interval(0, infinity),** e > 0 são propriedades equivalentes. Similarmente, **Type::Odd** é equivalente a **Type::Residue (1,2).** Há também membros da biblioteca **Type** que não correspondem à propriedades matemáticas e não podem ser usados em um comando **assume**, por exemplo **Type::PolyOf ou Type::Series**.

Nós ilustramos agora cada dos vários tipos de propriedades com um exemplo curto. A equação $(x^a)^b = x^{ab}$ não é geralmente válida, como o exemplo $x = -1$, $a = -2$, e $b = 1/2$ mostra. Contudo, isso é válido se b é uma integral:

```
>> assume(b, Type::Integer): (x^a)^b
    x^(a b)
>> unassume(b): (x^a)^b
    (x^a)^b
```

A função **linalg::isPosDef** checa se uma matriz é positiva definida. Para uma matriz com entradas simbólicas, isso pode não ser possível para decidir isso corretamente:

basic number domains	`Type::Complex`	$\mathbb{C}$
	`Type::Even`	$2\mathbb{Z}$
	`Type::Imaginary`	$\mathbb{R}i$
	`Type::Integer`	$\mathbb{Z}$
	`Type::Negative`	$\mathbb{R}_{<0}$
	`Type::NegInt`	$\mathbb{Z}_{<0}$
	`Type::NegRat`	$\mathbb{Q}_{<0}$
	`Type::NonNegative`	$\mathbb{R}_{\geq 0}$
	`Type::NonNegInt`	$\mathbb{N}$
	`Type::NonNegRat`	$\mathbb{Q}_{\geq 0}$
	`Type::NonZero`	$\mathbb{C} \setminus \{0\}$
	`Type::Odd`	$2\mathbb{Z}+1$
	`Type::PosInt`	$\mathbb{N}_{>0}$
	`Type::Positive`	$\mathbb{R}_{>0}$
	`Type::PosRat`	$\mathbb{Q}_{>0}$
	`Type::Prime`	prime numbers
	`Type::Rational`	$\mathbb{Q}$
	`Type::Real`	$\mathbb{R}$
	`Type::Zero`	$\{0\}$
intervals	`Type::Interval(a, b, T)`	$\{x \in T: a < x < b\}$
	`Type::Interval([a], b, T)`	$\{x \in T: a \leq x < b\}$
	`Type::Interval(a, [b], T)`	$\{x \in T: a < x \leq b\}$
	`Type::Interval([a], [b], T)`	$\{x \in T: a \leq x \leq b\}$
		a, b: expressions
		T: basic number domain
residue classes	`Type::Residue(a, b)` or `b * Type::Integer + a`	$b\mathbb{Z}+a$
		a, b: integers
relations	`= b`	$\{b\}$
	`<> b`	$\mathbb{C} \setminus \{b\}$
	`< b`	$\mathbb{R}_{<b}$
	`<= b`	$\mathbb{R}_{\leq b}$
	`> b`	$\mathbb{R}_{>b}$
	`>= b`	$\mathbb{R}_{\geq b}$
		b: expression

```
>> A := matrix([[1/a, 1], [1, 1/a]])
```

$$\begin{pmatrix} \frac{1}{a} & 1 \\ 1 & \frac{1}{a} \end{pmatrix}$$

```
>> linalg::isPosDef(A)
   Error: cannot check whether matrix component is posit\
   ive [linalg::factorCholesky]
```

Com a suposição inicial que o parâmetro **a** ser positivo e menor que 1, MuPAD pode decidir que é matriz é definida positiva:

```
>> assume(a, Type::Interval(0, 1))
```

$(0, 1)$

```
>> linalg::isPosDef(A)
```

TRUE

Propriedades deste tipo podem ser especificadas no seguinte modo mais intuitivo também:

```
>> assume(0 < a < 1)
```

$(0, 1)$

A função **simplify** reage às propriedades:

```
>> assume(k, Type::Residue(3,4))
```

$4\mathbb{Z}+3$

```
>> sin(k*PI/2)
```

$$\sin\left(\frac{\pi\, k}{2}\right)$$

```
>> simplify(%)
```

-1

A propriedade acima pode também ser especificada na seguinte forma equivalente:

```
>> assume(k, 4*Type::Integer + 3)
```

$4\mathbb{Z}+3$

As funções **Re**, **IM**, **sign**, e **abs** levam propriedades em conta:

```
>> assume(x > 1):
   Re(x*(x - 1)), sign(x*(x - 1)), abs(x*(x - 1))
```

$x\,(x-1)$, 1, $x\,(x-1)$

Desde que apenas uma quantidade limitada de propriedades matemáticas e regras de derivação estão implementadas no MuPAD, o sistema executa algumas simplificações quando avalia as propriedades de uma expressão não trivial. Assim a resposta de **getprop** ou **is** ´-e as vezes não “tão fechadas quanto possível.” Por exemplo, se x é um número real, então $x2 – x \geq –1/4$, mas **getprop** rende a seguinte falta de resposta precisa:

```
>> assume(x, Type::Real): getprop(x^2 - x)
    ℝ
```

Em adição com as propriedades dos identificadores individuais, ali existe uma *propriedade global* que é fixada para todos os identificadores em adição a seus individuais. Esta propriedade global pode ser formada e examinada via o especial identificador **Global**. Por exemplo, o seguinte fixa a propriedade da representação de um número real positivo para todos os identificadores sem um valor:

```
>> assume(Global, Type::Positive):
```

O mesmo pode ser alcançado com o comando **assume (Global > 0).** Agora, todo identificador tem pelo menos essa propriedade global, desde que se não a propriedade individual é fixada a isso:

```
>> unassume(x): is(x > 0)
    TRUE
```

Se um identificador já tem uma propriedade individual, então **getprop** e **is** trabalham com um "fim" lógico de global e a propriedade individual:

```
>> assume(x, Type::Integer): getprop(x)
    ℤ ∩ (0, ∞)
```

O comando **getprop (Global)** retorna a propriedade global:

```
>> getprop(Global)
    (0, ∞)
```

Você pode deletar a propriedade global via **unassume**. Se nenhuma propriedade global está posta, então **getprop (Global)** retorna **Global**:

```
>> unassume(Global): getprop(Global)
    Global
```

Em muitos casos, o argumento **Global** pode ser omitido. Se **prop** é qualquer propriedade, **assume (prop)** joga a propriedade global para **prop.** A chamada **is** (**prop)** checa se uma propriedade global implica a propriedade **prop.** A chamada **getprop()** retorna a propriedade global. A chamada **unassume** () deleta a propriedade global.

Exercício 9.5: Use MuPAD para mostrar:

$$\lim_{x\to\infty} x^a = \begin{cases} \infty & \text{for } a > 0\,, \\ 1 & \text{for } a = 0\,, \\ 0 & \text{for } a < 0\,. \end{cases}$$

Dica: Use a função **assume** para distinguir os casos.

Capítulo 10
Chance e probabilidade

Você pode usar os geradores de número aleatório do MuPAD **random**, **frandom** e **stats::xxxRandom** para executar muitos experimentos no MuPAD.

A chamada **random ()** gera um número aleatório não negativo de 12 dígitos. Você obtém uma sequência de 4 números aleatórios tais c Omo se seguem:

```
>> random(), random(), random(), random()
    427419669081, 321110693270, 343633073697, 474256143563
```

Se você quer gerar integrais aleatórias de um alcance diferente, você pode construir um gerador de número aleatório **generator:=random(m..n).** Você chama este gerador sem argumentos, e ele retorna integrais entre *m* e *n*. A chamada **random (n)** é equivalente a **random (0..n-1).** Assim você pode simular 15 rolos de um dado como se segue:

```
>> die := random(1..6):
>> dieExperiment := [die() $ i = 1..15]
    [5, 3, 6, 3, 2, 2, 2, 4, 4, 3, 3, 2, 1, 4, 4]
```

Nós damos ênfase a que você tem que especificar uma variável loop quando usa o gerador de sequência **$**, desde que do contrário **die()** é chamado apenas após e uma sequencia de cópias deste valor é gerado:

```
>> die() $ 15
    6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6
```

Aqui é uma simulação de 8 lances de moeda:

```
>> coin := random(2):
>> coinTosses := [coin() $ i = 1..8]
    [0, 0, 0, 1, 1, 1, 0, 0]
>> subs(coinTosses, [0 = head, 1 = tail])
    [head, head, head, tail, tail, tail, head, head]
```

O seguinte exemplo gera uniformemente distribuídos números de pontos flutuantes do intervalo [0,1]. Ele usa o gerador aleatório **frandom**:

```
>> randomNumbers := [frandom() $ i = 1..10]
   [0.2703567032, 0.8142678572, 0.1145977439,

      0.247668289, 0.436855213, 0.7507294917,

      0.5143284818, 0.47002619, 0.06956333824,

      0.5063265159]
```

A biblioteca **stats** cumpre funções para análise estatística. Você obtém informação entrando **info (stats)** ou **?stats**. Por exemplo, a função **stats::mean** co9mputa o valor principal $X = \frac{1}{n}\sum_{i=1}^{n} xi$ de uma lista de números $[x_1,...,x_n]$:

```
>> stats::mean(dieExperiment),
   stats::mean(coinTosses),
   stats::mean(randomNumbers)
    16/5, 3/8, 0.4194719824
```

A função **stats::variance** retorna a discrepância

$$V = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - X)^2 \ :$$

```
>> stats::variance(dieExperiment),
   stats::variance(coinTosses),
   stats::stdev(dieExperiment),
   stats::stdev(coinTosses),
   stats::stdev(randomNumbers)
    sqrt(61)*sqrt(35)/35, sqrt(15)*sqrt(14)/28, 0.2476850434
```

Se você especificar a opção **Population**, o sistema retorna $\sqrt{n - \frac{1}{n}V}$ ao invés disso:

```
>> stats::stdev(dieExperiment, Population),
   stats::stdev(coinTosses, Population),
   stats::stdev(randomNumbers, Population)
    sqrt(122)*sqrt(3)/15, sqrt(15)/8, 0.2349746638
```

A estrutura de dado **Dom::Multiset** (veja **? Dom::Multiset)** provê uns simples sentidos de determinação de frequências em sequências. A chamada **Dom::Multiset (a, b,...)** retorna um multi com junto, o qual é imprimido como um conjunto de listas. A primeira entrada de cada tal lista é um dos argumentos; a segunda entrada conta o número de ocorrências no argumento sequência:

```
>> Dom::Multiset(a, b, a, c, b, b, a, a, c, d, e, d)
    {[a, 4], [b, 3], [c, 2], [d, 2], [e, 1]}
```

Se você simular 1000 rolos de um dado, você pode obter as seguintes frequências:

```
>> rolls := die() $ i = 1..1000:
>> Dom::Multiset(rolls)
    {[2, 152], [1, 158], [3, 164], [6, 162], [5, 176], [4, 188]}
```

Neste caso, nós vamos ter rolado 158 vezes uma 1, 152 vezes uma 2 etc.

Um exemplo da teoria dos números é a distribuição do maior divisor comum (mdc) de pares aleatórios de integrais. Nós usamos a função **zip** (Seção 4.6) para combinar duas listas aleatórias via a função **igcd**, a qual computa o mdc das integrais:

```
>> list1 := [random() $ i=1..1000]:
>> list2 := [random() $ i=1..1000]:
>> gcdlist := zip(list1, list2, igcd)
   [1, 7, 1, 1, 1, 5, 1, 3, 1, 1, 1, 3, 6, 1, 3, 5, ...]
```

Agora, nós usamos **Dom::Multiset** para contar a frequência dos mdcs individuais:

```
>> frequencies := Dom::Multiset(op(gcdlist))
   {[11, 5], [13, 3], [14, 2], ..., [377, 1], [1, 596]}
```

O resultado se torna muito mais legível se nós classificarmos ele pela primeira entrada das sublistas. Nós empregamos a função **sort** que leva uma função representando uma ordem classificando como segundo argumento. Esta função decide quais dos dois elementos *x, y* podem preceder o outro. Nós nos referimos a página de ajuda correspondente: **?sort**. Nesse caso, *x, y* são listas com duas entradas, e nós queremos *x* para aparecer antes de *y* se nós temos $x[1] < y[1]$ (por exemplo, nós classificamos numericamente com respeito as primeiras entradas):

```
>> sortingOrder := (x, y) -> (x[1] < y[1]):
>> sort([op(frequencies)], sortingOrder)
   [[1, 596], [2, 142], [3, 84], ..., [212, 1], [377, 1]]
```

Nesse experimento, 596 saíram dos 1000 pares aleatórios escolhidos tiveram um mdc de 1 e portanto são coprimos. Assim, nós encontramos 59.6% como uma aproximação da probabilidade que duas escolhas aleatoriamente escolhidas de integrais são coprimos. O valor teórico desta probabilidade é $6/\pi^2 \approx 0.6079$. . =^ 60.79%.

A biblioteca **stats** provê muitas funções de distribuição estocástica. Cada tal distribuição **xxx**, diz, consiste de quatro funções: uma distribuição cumulativa **xxxCDF**, uma função de probabilidade de densidade **xxxPDF** (ou uma função de probabilidade discreta **xxxPF**, respectivamente), uma função quantil **xxxQuantile**, e um gerador de número aleatório **xxxRandom**. Por exemplo, o seguinte comando cria uma lista de números aleatórios distribuídos de acordo com a distribuição normal padrão com sentido 0 e a variância 1:

```
>> randomGenerator := stats::normalRandom(0, 1):
   data := [randomGenerator() $ i = 1..1000]:
```

A biblioteca **stats** inclui uma implementação do clássico $X2$ – *teste*. Nós usamos isso aqui para testar se um dado aleatório gerado acima faz ao invés disso satisfaz uma distribuição normal. Pretendendo que nós não sabemos o significado e a variância do dado, nós computamos estimativas estatísticas desses parâmetros:

```
>> m := stats::mean(data)
    0.02413100072
>> V := stats::variance(data)
    1.057017094
```

O $X2$ – *teste* requer especificar uma "partição celular" da linha real para comparar as frequências observadas do dado caindo dentro das células com as frequências esperadas dando uma distribuição hipotética dos dados. A função **stats::equiprobableCells** é uma função de utilidade conveniente para computar uma partição celular consistindo de células que são equiprováveis com respeito a uma dada distribuição. A seguinte chamada de partições tem uma linha real dentro de 32 células que são equiprováveis com respeito à distribuição normal com o sentido empírico e variância coimputada acima:

```
>> cells := stats::equiprobableCells(32,
                      stats::normalQuantile(m, V))
   [[-infinity, -1.89096853], [-1.89096853, -1.553118836],

       ..., [1.939230531, infinity]]
```

O teste $X2$ – bondade – de – ajuste é implementado por **stats::csGOFT**. Nós usamos ele p0ara testar quão bom o dado aleatório ajusta uma distribuição normal com sentido e variância dada pelos valores empíricos **m** e **V**:

```
>> stats::csGOFT(data, cells,
                 CDF = stats::normalCDF(m, V))
   [PValue = 0.9202941502, StatValue = 20.67199999,

      MinimalExpectedCellFrequency = 31.24999998]
```

O primeiro valor retornado por **stats::csGOFT** é o nível significativo atingido pelo dado. Desde que este valor não é pequeno, o dado dado pessa pelo teste tão bem. Veja a página de ajuda de **stats::csGOFT** para detalhes futuros.

Finalmente, nós caducamos os dados pela adição de 35 zeros:

```
>> data := append(data, 0 $ 35):
```

Nós checamos se um novo dado ainda pode ser considerado como uma amostra normalmente distribuída:

```
>> m := stats::mean(data): V := stats::variance(data):
   cells := stats::equiprobableCells(32,
                      stats::normalQuantile(m, V)):
   stats::csGOFT(data, cells,
                 CDF = stats::normalCDF(m, V))
   [PValue = 0.001078732853, StatValue = 60.82222221,

      MinimalExpectedCellFrequency = 32.34374998]
```

Agora, o nível atingido de significância 0.0010... indica que a hipótese de uma distribuição normal de dado deve ser rejeitada em níveis significativos tão baixos como 0.001.

Exercício 10.1: Três dados são lançados simultaneamente. Para cada valor entre 3 e 18, a seguinte tabela contém as frequências esperadas do total de pontuação dos dados quando rolando o dado 216 vezes:

score

3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	3	6	10	15	21	25	27	27	25	21	15	10	6	3	1

frequency

Simule 216 lances e compare as frequências que você observa com esses da tabela.

Exercício 10.2: O método Monte – Carlo para aproximação da área de uma região $A \subset \mathbb{R}2$ trabalha como se segue. Primeiro, nós escolhemos um (preferencialmente pequeno) retângulo Q incluindo A. Então nós escolhemos aleatoriamente n pontos em Q. Se m desses pontos pertencem em A, a seguinte estimativa mantém para suficientemente grande n:

Área de $A \approx \frac{m}{n} x$ *area de* Q.

Deixe **r ( )** ser uma chamada para um gerador **r** de numerosa aleatórios uniformemente distribuídos no intervalo [0, 1]. Nós podemos usar **r** para gerar vetores aleatórios uniformemente distribuídos no retângulo Q = [0, a] x [0, b] via **[a*r(), b*r()].**

a) Considere o quadrante a direita superior do círculo de unidade em volta da origem. Use simulação Monte-Carlo com Q = [0,1] x [0,1] para aproximar essa área. Este modo, um consegue aproximações aleatórias de π.

b) Deixe $f : x \mapsto x \sin(x) + \cos(x) \exp(x)$. Determine uma aproximação para $\int_0^1 f(x)dx$. Para este propósito, encontre um salto acima de M para f no intervalo [0,1] e aplique a simulação para Q = [0,1] x [0, M]. Compare seus resultados para a integral exata.

Capítulo 11 Gráficos
Com o lançamento do MuPAD 3.0, os gráficos do sistema foram reescritos. Nós nos referimos para o **plot document** para uma introdução aos novos gráficos.

Capítulo 12 O mecanismo de história

Toda entrada para MuPAD rende um resultado após avaliação pelo sistema. Os objetos computados são armazenados internamente em uma *tabela história*. Note que o resultado de cada alegação é guardado, até mesmo se ele não é impresso na tela. Você pode usar isso depois pelos sentidos da função **last**. O comando **last (1)** retorna o resultado anterior, last **(2)** o último mas um, e assim em. Ao invés de **last (i)**, você pode usar a notação abreviada %i. Além do mais, **%** é um atalho para **%1** ou **last(1)**. Assim a entrada

```
>> f := diff(ln(ln(x)), x): int(f, x)
```

Pode ser passada para o sistema na seguinte forma equivalente:

```
>> diff(ln(ln(x)), x): int(%, x)
```

Isso habilita você a acessar resultados intermediários que não tem sido nomeados para um identificador. Isso é notável que o uso de **last** pode acelerar certas avaliações quando comparadas ao uso de identificadores para armazenar resultados intermediários. No seguinte exemplo, nós tentamos primeiro computar uma integral definida simbólicamente. Após reconhecer que MuPAD não computa um valor simbólico, nós perguntamos para uma aproximação de pontos flutuantes:

```
>> f := int(sin(x)*exp(x^3)+x^2*cos(exp(x)), x=0..1)
```

$$\int_0^1 x^2 \cos\left(e^x\right) + \sin\left(x\right)\, e^{x^3}\, dx$$

```
>> startingTime := time():
   float(f);
   (time() - startingTime)*msec
```

0.5356260737

750 msec

A função **time** retorna o total de tempo computando (em milissegundos) usados pelo sistema depois do começo da sessão. Assim a diferença impressa é o tempo para computar a aproximação dos pontos flutuantes. Neste exemplo, nós podemos reduzir o tempo de computação dramaticamente pelo emprego de **last**:

```
>> f := int(sin(x)*exp(x^3)+x^2*cos(exp(x)), x = 0..1)
```

$$\int_0^1 x^2 \cos\left(e^x\right) + \sin\left(x\right)\, e^{x^3}\, dx$$

```
>> startingTime := time():
   float(%2);
   (time() - startingTime)*msec
```

0.5356260737

40 msec

Neste caso, a razão para o ganho em velocidade é que MuPAD não *reavalia* o objeto que **last(i), %i**, ou **%** se refere. Assim chamadas para a forma **last** formam uma exceção para a avaliação completa usual em nível interativo (Seção 5.2):

```
>> delete x: sin(x): x := 0: %2
```

$\sin(x)$

Você pode fazer cumprir avaliação completa usando **eval**:

```
>> delete x: sin(x): x := 0: eval(%2)
```

0

Por favor, note que o valor de **last (i)** pode diferir do iº mas última saída *visível* se você tem supresso a saída da tela de alguns resultados intermediários terminando os comandos correspondentes co uma coluna. Também note que o valor da expressão **last (i)** muda permanentemente durante a computação:

```
>> 1: last(1) + 1; last(1) + 1
```

2

3

O ambiente variável **HISTORY** determina o número de resultados que MuPAD guarda em uma sessão e que pode ser acessados via **last**:

```
>> HISTORY
     20
```

Este padrão significa que MuPAD armazena as 20 expressões prévias. Com certeza você pode mudar este padrão nomeando um valor diferente para **HISTORY**. Isso pode ser apropriado quando MuPAD tem que controlar altos objetos (tais como matrizes muito extensas) que enchem-se de uma parte significante da memória de seu computador. Cópias desses objetos são armazenadas em uma tabela história, requerendo espaço de armazenamento adicional. Neste caso, você reduziria o carregamento de memória escolhendo valores pequenos em **HISTORY**. Note que **HISTORY** apenas rende o valor do interativo "profundidade história". Dentro de um procedimento, **last** apenas aceita os argumentos 1, 2 e 3.

Nós recomendamos fortemente usar **last** apenas interativamente. O uso de **last** sem procedimentos é considerado um estilo mau de programação e deve ser evitado.

Capítulo 13 Entrada e Saída

13.1 Saídas de expressões

MuPAD não mostra todos os resultados computados na tela. Exemplos típicos são os comandos dentro dos loops **for** (Capoítulo 16) ou procedimentos (Capítulo 18): apenas o resultado final (por exemplo, o resultado do último comando) é impresso e a saída dos resultados intermediários é suprimida. Não obstante, você pode deixar MuPAD imprimir resultados intermediários ou mudar o formato da saída.

13.1.1 Imprimindo expressões na tela

A função **print** saí objetos do MuPAD na tela:

```
>> for i from 4 to 5 do
     print("The ", i, "th prime is ", ithprime(i))
   end_for
                "The ", 4, "th prime is ", 7

               "The ", 5, "th prime is ", 11
```

Nós rechamamos que **ithprime (i)** computa o iº número primo. MuPAD inclui o texto em dupla citações. Use a opção **Unquoted** para suprimir isto:

```
>> for i from 4 to 5 do
     print(Unquoted,
           "The ", i, "th prime is ", ithprime(i))
   end_for
                 The , 4, th prime is , 7

                The , 5, th prime is , 11
```

Mais futuramente, nós podemos eliminar as vírgulas da saída pelos sentidos das ferramentas para manipular fileiras que são apresentadas na Seção 4.11:

```
>> for i from 4 to 5 do
     print(Unquoted,
           "The " . expr2text(i) . "th prime is " .
           expr2text(ithprime(i)) . ".")
   end_for
                   The 4th prime is 7.

                   The 5th prime is 11.
```

Aqui, a função **expr2text** é usada para converter o valor de i e o número primo retornada por ithprime (i) para cordas. Então, o operador concatenação **.** combina eles com as outras cordas para uma única corda.

Alternativamente você pode usar a função **fprint**, a qual escreve dados para um arquivo ou na tela. Em contraste com **print**, isso não faz sair seus argumentos como expressões individuais. Ao invés disso, **fprint** combina eles para uma única corda (se você usar a opção **Unquoted**):

```
>> a := one: b := string:
>> fprint(Unquoted, 0, "This is ", a, " ", b)
   This is one string
```

O segundo argumento 0 diz **fprint** para direcionar esta saída para a tela. Note que **fprint** não usa a "bela impressora" (a qual é explicada abaixo).

13.1.2 Modificando o formato da saída

Algumas versões do MuPAD apresentam um caderno ambiente que pode imprimir resultados do MuPAD com uma tipografia especial para expressões matemáticas ("expressões tipográficas"): sinais integrais são mostrados como $\int$, somas simbólicas aparecem como $\sum$ etc. Isto é o formato usado na maioria das "regiões de resultados" deste livro. Há dois outros formatos de saída disponíveis em todas as versões do MuPAD, das quais são também usadas em notebooks para largas saídas, desde como desta escrita formatação de expressões não são quebradas sobre várias linhas. O seguinte refere-se para esta saída baseada em ASCII apenas. Em versões do MuPAD com expressões compostas, o seguinte somente é válido se a formatação está desativada via o correspondente menu da interface do notebook.

Sem formatação, MuPAD normalmente imprime expressões em uma forma bidimensional com simples (ASCII) caracteres:

```
>> diff(sin(x)/cos(x), x)
                          2
                      sin(x)
                      ------- + 1
                          2
                      cos(x)
```

Este formato é conhecido como *pretty print*. Isso parece-se a notação matemática usual. Então, isso é muitas vezes mais fácil de ler que uma única linha de saída. Contudo, MuPAD apenas usa pretty print para saída e não é válido para formato de entrada: em uma interface gráfica de usuário você não pode copiar alguns textos de saída com o mouse e copiá-los como entrada do MuPAD em outro lugar.

A variável ambiente **PRETTYPRINT** controla o formato de saída. O valor padrão da variável é **TRUE**, por exemplo, o formato pretty print é usado para a saída. Se você joga esta variável para **FALSE**, você obtém uma saída unidimensional de forma que pode também ser usada como entrada:

```
>> PRETTYPRINT := FALSE: diff(sin(x)/cos(x), x)
   1/cos(x)^2*sin(x)^2 + 1
```

Se uma saída excederia o comprimento da linha, o sistema automaticamente quebra as linhas:

```
>> PRETTYPRINT := TRUE: taylor(sin(x), x = 0, 16)
       3     5     7       9         11
      x     x     x       x         x
  x - -- + --- - ---- + ------ - -------- +
      6    120   5040   362880   39916800

       13            15
      x             x             17
  ---------- - ------------- + O(x  )
  6227020800   1307674368000
```

Você pode jogar a variável ambiente **TEXTWIDTH** para o comprimento de linha desejado. O valor padrão é 75 (caracteres), e você pode nomear qualquer integral entre 10 e $2^{31} - 1$ para isto. Por exemplo, se você computar $(\ln \ln x)^n$, então você obtém a seguinte saída:

```
>> diff(ln(ln(x)), x, x)
                        1         1
                  - -------- - ---------
                      2          2      2
                     x  ln(x)   x  ln(x)
```

Se você reduzir o valor de **TEXTWIDTH**, o sistema quebra a saída através de duas linhas:

```
>> TEXTWIDTH := 20: diff(ln(ln(x)),x,x)
       1
  - -------- -
     2
    x  ln(x)

        1
    ---------
     2      2
    x  ln(x)
```

O valor padrão é restaurado deletando **TEXTWIDTH**:

```
>> delete TEXTWIDTH:
```

Você também pode controlar a saída pelas preferências de usuário definidas. Isso é discutido na Capítulo 14.1.

13.2 Lendo e escrevendo arquivos

Você pode salvar os valores de identificadores ou uma sessão completa do MuPAD para um arquivo e ler o arquivo depois dentro de outra sessão do MuPAD.

13.2.1 As funções write e read

A função **write** armazena os valores dos identificadores em um arquivo, assim que você pode reutilizar os resultados computados em outra sessão do MuPAD. No seguinte exemplo, nós salvamos os valores dos identificadores **a** e **b** para o arquivo **ab.mb**:

```
>> a := 2/3: b := diff(sin(cos(x)), x):
>> write("ab.mb", a, b)
```

Você passa o nome do arquivo como uma corda (Sessão 4.11) incluído em dupla citações ‘’. O sistema então cria um arquivo com o nome (sem ‘’). Se você ler esse arquivo dentro de outra sessão no MuPAD via a função **read**, você pode acessar os valores dos identificadores **a** e **b** sem computá-los novamente:

```
>> a := 2/3: b := diff(sin(cos(x)), x):
>> write(Text, "ab.mu", a, b)
```

```
>> reset(): read("ab.mb"): a, b
```

$$\frac{2}{3},\ -\sin(x)\cos(\cos(x))$$

Se você usar a função **write** como no exemplo acima, isso cria um arquivo no formato binário específico no MuPAD. Por convenção, um arquivo neste formato deve ter a extensão de arquivo chamada "**.mb**." Você pode chamar a função **write** com a opção **Text**. Isso gera um arquivo em um formato de texto legível:

O arquivo **ab.mu** agora contém os seguintes dois comandos do MuPAD sintaticamente corretos:

```
a := hold(2/3):
b := hold(-sin(x)*cos(cos(x))):
```

Você pode usar a função **read** para ler esse arquivo:

```
>> a := 1: b := 2: read("ab.mu"): a, b
```

$$\frac{2}{3},\ -\sin(x)\cos(\cos(x))$$

O formato de arquivos de texto gerado por **write** contém comandos válidos no MuPAD. Certamente, você pode usar qualquer editor para gerar tal arquivo de texto "manualmente" e ler ele dentro de uma sessão do MuPAD. De fato isso é um modo natural para prosseguir quando você desenvolve procedimentos mais complexos do MuPAD.

13.2.2 Salvando uma sessão do MuPAD

Se você chamar a função **write** sem fornecer quaisquer identificadores como argumentos, o sistema escreve os valores de *todos* os identificadores tendo um valor para um arquivo. Assim, é possível restaurar o estado da atual sessão via **read** em um tempo depois:

```
>> result1 := ...; result2 := ...; ...
>> write("results.mb")
```

A função **protocol** grava as entradas e as saídas da tela de uma sessão em um arquivo. Entradas e saídas são escritos para este arquivo até você entrar com **protocol ( )**:

```
>> protocol("logfile"):
>> limit(sin(x)/x, x = 0)
    1
>> protocol():
```

Esses comandos geram um arquivo de texto chamado **logfile** com os seguintes conteúdos:

```
limit(sin(x)/x, x = 0)

                        1

protocol():
```

Não é possível ler um arquivo gerado por **protocol** denjtro de uma sessão do MuPAD.

13.2.3 Lendo dados de um arquivo de texto

Muitas vezes você quer usar dados em MuPAD que são gerados por um diferente software (por exemplo, você pode querer ler em valores estatísticos para futuro processamento), ou acessar todos os arquivos em, algum diretório automaticamente. Isso é possível com a ajuda da função **import::readdata** da biblioteca **import**. Esta função converte o conteúdo de um arquivo para uma lista aninhada do MuPAD. Você pode considerar o arqui9vo como uma "matriz" com quebras de linha indicando o começo de uma nova fileira. Note que as fileiras podem ter comprimentos diferentes. Você pode passar um caractere arbitrário para **import::readdata** como um separador de coluna.

Suponha que você tenha um arquivo **numericalData** com as seguintes 4 linhas:

```
1   1.2   12
2.34  234
  34  345.6
4    44      444
```

Por padrão, caracteres em branco são assumidos como separadores de coluna. Assim você pode ler este arquivo dentro de uma sessão do MuPAD como se segue:

```
>> data := import::readdata("numericalData"):
>> data[1]; data[2]; data[3]; data[4]
```

$$[1, 1.2, 12]$$

$$[2.34, 234]$$

$$[34, 345.6]$$

$$[4, 44, 444]$$

A página de ajuda para **import::readdata** provê futura informação.

Capítulo 14 Utilidades

Neste capítulo nós apresentamos algumas funções úteis. Devido as limitações de espaço, nós não explicamos sua completa funcionalidade e refere às páginas de ajuda para informação mais detalhada.

14.1 Preferências definidas pelo usuário

Você pode customizar o comportamento do MuPAD usando *preferências*. O seguinte comando listas todas as preferências:

```
>> Pref()
   Pref::alias           : TRUE
   Pref::ansi            : TRUE
   Pref::autoPlot        : FALSE
   Pref::callBack        : NIL
   Pref::callOnExit      : NIL
   Pref::dbgAutoList     : FALSE
   Pref::echo            : TRUE
   Pref::experimentalInt: FALSE
   Pref::floatFormat     : "g"
   Pref::ignoreNoDebug   : FALSE
   Pref::keepOrder       : DomainsOnly
   Pref::kernel          : [3, 1, 0]
   Pref::matrixSeparator: ", "
   Pref::maxMem          : 0
   Pref::maxTime         : 0
   Pref::noProcRemTab    : FALSE
   Pref::output          : NIL
   Pref::postInput       : NIL
   Pref::postOutput      : NIL
   Pref::prompt          : TRUE

   Pref::promptString    : ">> "
   Pref::report          : 0
   Pref::timesDot        : " "
   Pref::trailingZeroes  : FALSE
   Pref::typeCheck       : Interactive
   Pref::userOptions     : ""
   Pref::verboseRead     : 0
   Pref::warnChanges     : FALSE
   Pref::warnDeadProcEnv: FALSE
   Pref::warnLexProcEnv  : FALSE
```

Nós referimos para a página de ajuda **?Pref** para uma descrição completa de todas as preferências. Apenas umas poucas opções são discutidas abaixo.

Você pode usar a preferência **report** para requisitar informação regular na memória alocada do MuPAD, a memória realmente usada, e o tempo de computação decorrido. Argumentos válidos para **report** são integrais entre 0 e 9. O valor padrão 0 significa que nenhuma informação é mostrada. Se você escolher o valor 9, você permanentemente obtém informação sobre o atual estado do MuPAD.

```
>> Pref::report(9): int(sin(x)^2/x, x = 0..1)
   [used=2880k, reserved=3412k, seconds=1]
   [used=3955k, reserved=4461k, seconds=1]
   [used=7580k, reserved=8203k, seconds=2]

     γ   Ci(2)   ln(2)
     ─ - ──── + ────
     2    2      2
```

A preferência **floatFormat** controla a saída dos números de pontos flutuantes. Por exemplo, se você fornece o argumento **"e"**, números de pontos flutuantes são impressos com mantissa e exponencial (por exemplo, **1.234e-7** = $1.234 \cdot 10^{-7}$). Se você usar o argumento **"f"**, MuPAD imprime eles em ponto de representação fixada:

```
>> Pref::floatFormat("e"): float(exp(-50))
     1.928749848 · 10^-22
>> Pref::floatFormat("f"): float(exp(-50))
     0.000000000000000000001928749848
```

Você pode usar preferências para controlar a saída na tela em outras formas também. Por exemplo, depois de chamar **Pref::output (F)**, MuPAD passa todo resultado computado pelo kernel para a função **F** antes de imprimir ele na tela. A saída na trela é então o resultado da função **F** ao invés do resultado originalmente computado pelo kernel. No seguinte exemplo, nós usamos isto para computar e sair a normalização de toda expressão requisitada. Nós definimos um procedimento **F** (Capítulo 18) e passamos ele para **Pref::output**:

```
>> Pref::output(x -> (x, normal(x))):
>> 1 + x/(x + 1) - 2/x
     x   2          -2 x^2 + x + 2
     ─ - ─ + 1, - ─────────────────
     x+1 x           x^2 + x
```

A biblioteca **generate** contém funções para conversão de expressões do MuPAD para o formato de entrada de outras linguagens de programação (C, Fortran, ou TEX).No seguinte exemplo, a saída do MuPAD é convertida à uma corda. Você pode então escrever essa corda para um arquivo de texto para futuro processamento com TEX:

```
>> Pref::output(generate::TeX): diff(f(x),x)
     "\frac{\partial}{\partial x} f\left(x\right)"
```

O seguinte comando reseta a saída rotina para seu estado original:

```
>> Pref::output(NIL):
```

Alguns usuários querem obter informação em certas características de todas as computações, tais como os tempos computados. Isso pode ser alcançado com as funções **Pref::postInput** e **Pref::postOutput**. Ambos levam procedimentos do MuPAD como argumentos, os quais são então chamados depois de cada entrada e saída, respectivamente. No seguinte exemplo, nós usamos um procedimento que nomeia o tempo do sistema retornado por **time** ( ) para o identificador global **Time**. Isso começa um "timer" antes de cada computação:

```
>> Pref::postInput(proc() begin Time := time() end_proc):
```

Nós definimos um procedimento **myInformation** o qual – entre outras coisas – usa o timer para determinar o tempo levado pela computação. Isso emprega **expr2text** (Seção 4.11) para converter o tempo de valor numérico para uma corda e concatena isso com algumas outras cordas. Além disso, o procedimento usa **domtype** para encontrar o tipo de domínio do objeto e converte ele para uma corda tão bem. Finalmente ele concatena a informação de tempo, alguns espaços em branco " ", e o informação de tipo via **_concat**. A função **length** é usada para determinar o número preciso de espaços em branco assim que o tipo de domínio aparece corado a direita na tela:

```
>> myInformation := proc() begin
       "domain type : ". expr2text(domtype(args()));
       "time : ". expr2text(time() - Time). " msec";
       _concat(%1,
               " " $ TEXTWIDTH-1-length(%1)-length(%2),
               %2)
   end_proc:
```

Nós passamos este procedimento como argumento para **Pref::postOutput**:

```
>> Pref::postOutput(myInformation):
```

Depois de cada resultado computado, o sistema agora imprime a corda gerada pelo procedimento **myInformation** na tela:

```
>> factor(x^3 - 1)
     (x - 1) · (x + x^2 + 1)

  time : 80 msec                    domain type : Factored
```

Você pode resetar uma preferência para seu valor padrão especificando **NIL** como argumento. Por exemplo, o comando **Pref::report (NIL)** reseta o valor de **Pref::report** para 0. Similarmente, **Pref (NIL)** reseta *todas* as preferências para seus valores padrão.

Exercício 14.1: A função do MuPAD **bytes** retorna a quantidade de memórias lógicas e físicas usadas pela sessão corrente do MuPAD. Deixe essa informação aparecer na tela depois de cada saída, usando **Pref::postOutput**.

14.2 Informação nos algoritmos do MuPAD

Algumas das funções do sistema do MuPAD podem prover informação de tempo corrido. O seguinte comando faz todos tais procedimentos produzirem informação adicional na tela:

```
>> setuserinfo(Any, 1):
```

Como um exemplo, nós invertemos a seguinte matriz (Seção 4.15) sobre o anel de integrais de modulo 11:

```
>> M := Dom::Matrix(Dom::IntegerMod(11)):
   A := M([[1, 2, 3], [2, 4, 7], [0, 7, 5]]):
   A^(-1)
  Info: using Gaussian elimination (LR decomposition)

    ⎛ 1 mod 11   0 mod 11   6 mod 11 ⎞
    ⎜ 3 mod 11   4 mod 11   8 mod 11 ⎟
    ⎝ 9 mod 11   1 mod 11   0 mod 11 ⎠
```

Você obtém informações mais detalhadas acrescentando o segundo argumento de **seturseinfo** (o "nível de informação"):

```
>> setuserinfo(Any, 3): A^(-1)
  Info: using Gaussian elimination (LR decomposition)
  Info: searching for pivot element in column 1
  Info: choosing pivot element 2 mod 11 (row 2)
  Info: searching for pivot element in column 2
  Info: choosing pivot element 7 mod 11 (row 3)
  Info: searching for pivot element in column 3
  Info: choosing pivot element 5 mod 11 (row 3)

    ⎛ 1 mod 11   0 mod 11   6 mod 11 ⎞
    ⎜ 3 mod 11   4 mod 11   8 mod 11 ⎟
    ⎝ 9 mod 11   1 mod 11   0 mod 11 ⎠
```

Se você entrar

```
>> setuserinfo(Any, 0):
```
the system stops printing additional information:

```
>> A^(-1)
```

$$\begin{pmatrix} 1 \bmod 11 & 0 \bmod 11 & 6 \bmod 11 \\ 3 \bmod 11 & 4 \bmod 11 & 8 \bmod 11 \\ 9 \bmod 11 & 1 \bmod 11 & 0 \bmod 11 \end{pmatrix}$$

O primeiro argumento de **setuserinfo** pode ser um nome de procedimento arbitrário ou nome de biblioteca. Então os correspondentes procedimentos(s) provê informação adicional. Programadores das funções do sistema construíram comandos de saída dentro do código via **userinfo**. Esses comandos podem usar isso nosa seus próprios procedimentos tão bem (**?userinfo**).

14.3 Reiniciando uma sessão do MuPAD

O comando **reset** ( ) reseta uma sessão do MuPAD para seu estado inicial. Depois de tudo, todos os identificadores que você definiu provisoriamente não tem nenhum valor e todas as variáveis ambientes são resetadas para seus valores padrões:

```
>> a := hello: DIGITS := 100: reset(): a, DIGITS
   a, 10
```

14.4 Executando comandos do sistema de operação

Você pode usar a função **system** (ou o símbolo de exclamação ! para atalho) para executar um comando de sistema de operação. Em plataformas UNIX, o seguinte comando lista os conteúdos do atual diretório:

```
>> !ls
   changes/    demo/  examples/   mmg/   xview/   bin/
   copyright/  doc/   lib/        tex/
```

Você pode nem usar a saída de tal um comando para futura computação nem salvá-lo para um arquivo. Nas versões terminais do MuPAD em UNIX/Linux, **system** retorna o status de erro do sistema operacional para a sessão do MuPAD.
A função **system** não está disponível em todas as plataformas. Por exemplo, você não pode nem usar ela em um sistema Windows nem em um Macintosh.

Capítulo 15 Tipos de especificadores

A estrutura de dados de um objeto do MuPAD é seu tipo de domínio, o qual pode ser requerido pelos sentidos da função **domtype**. O tipo de domínio reflete a estrutura que o kernel do MuPAD usa internamente para administrar os objetos. O tipo conceito também leva a classificação dos objetos de acordo com seus significados matemáticos: número, conjuntos, expressões, expansões de séries, polinômios, etc.

Nessa seção, nós descrevemos como obter informação detalhada sobre a estrutura matemática dos objetos. Por exemplo, como você descobre eficientemente se uma integral do tipo de domínio **DOM_INT** é *positiva* ou *par*, ou se todos os elementos de um conjunto são equações?

Tais tipos conferem estar apenas relevante quando usando MuPAD interativamente: você pode controlar o sentido matemático de um objeto direcionando inspeção. Checagem de tipo são principalmente usados para implementação de algoritmos matemáticos, por exemplo, quando programando procedimentos no MuPAD (capítulo 18). Por exemplo, um procedimento para diferenciação de expressões tem que decidir se sua entrada é um produto, uma composição de funções, uma chamada simbólica de uma função conhecida, etc. Cada caso requer uma ação diferente, tal como fornecer a regra do produto, a regra de série, etc.

15.1 As funções type e testtype

Para a maioria dos objetos do MuPAD, a função **type** retorna, como **domtype**, o tipo domínio:

```
>> type([a, b]), type({a, b}), type(array(1..1))
   DOM_LIST, DOM_SET, DOM_ARRAY
```

Para expressões do tipo de domínio **DOM_EXPR**, a função **type** rende uma fina distinção de acordo com o sentido matemático da expressão: somas, produtos, chamadas de funções, etc.:

```
>> type(a + b), type(a*b), type(a^b), type(a(b))
   "_plus", "_mult", "_power", "function"
>> type(a = b), type(a < b), type(a <= b)
   "_equal", "_less", "_leequal"
```

O resultado retornado por **type** é a função chamada que gera a expressão (internamente, uma soma simbólica ou produto é representado por uma chamada do sistema de funções **_plus** ou **_mult**, respectivamente). Mais geralmente, o resultado para uma chamada simbólica de um sistema de função é o identificador da função como uma corda:

```
>> type(ln(x)), type(diff(f(x), x)), type(fact(x))
   "ln", "diff", "fact"
```

Você pode usar ambos os tipos de domínio **DOM_INT**, **DOM_EXPR**, etc. e as cordas retornadas por **type**, como *tipo específicos*. Ali existe uma variedade de outros tipos de especificadores em adição à "digitação padrão" de objetos do MuPAD dados por **type.** Um exemplo é **Type::Numeric**. Este tipo cumpre todos os objetos "numéricos" (do tipo de domínio **DOM_INT**, **DOM_RAT**, **DOM_FLOAT**, ou **DOM_COMPLEX**).

A chamada **testtype (object, TypeSpecifier)** checa se um objeto queixa-se c om o tipo especificado. O resultado é ou **TRUE** ou **FALSE** Vários tipos de especificadores podem corresponder para um objeto:

```
>> testtype(2/3, DOM_RAT), testtype(2/3, Type::Numeric)
   TRUE, TRUE
>> testtype(2 + x, "_plus"), testtype(2 + x, DOM_EXPR)
   TRUE, TRUE
>> testtype(f(x), "function"), testtype(f(x), DOM_EXPR)
   TRUE, TRUE
```

Exercício 15.1: Considere a expressão

$$f(i) = \frac{i^{5/2} + i^2 - i^{1/2} - 1}{i^{5/2} + i^2 + 2\,i^{3/2} + 2\,i + i^{1/2} + 1}$$

Como o MuPAD decide se o conjunto

```
>> S := {f(i) $ i = -1000..-2} union {f(i) $ i=0..1000}:
```

Contém apenas números racionais? Dica: Para uma integral específica **i**, use a função **normal** para simplificar subexpressões de **f (i)** contendo raízes quadradas.

Exercício 15.2: Considere a expressão sin ($i\,\pi$/200) com valores integrais de i entre 0 e 100. Quais deles são simplificados pela função do MuPAD **sin**, pela qual é retroanda como valores simbólicos sin(.)?

15.2 Verificação de tipo confortável: a biblioteca de Type

Os especificadores de tipo apresentados acima são úteis apenas para checar estruturas relativamente simples. Para tipos de checagem mais avançados, tipos mais flexíveis de especificações são necessários. Por exemplo, como você pode checar sem inspeção direta se o objeto [1, 2, 3,...] é uma lista de integrais positivas?

Para este propósito, a biblioteca **Type** provê futuros tipos de especificadores e construtores, os quais são reconhecidos por **testtype**:

```
>> info(Type)
   Library 'Type': type expressions and properties

   -- Interface:
   Type::AlgebraicConstant, Type::AnyType,
   Type::Arithmetical,      Type::Boolean,
   Type::Complex,           Type::Constant,
   Type::ConstantIdents,    Type::Equation,
   Type::Even,              Type::Function,
   Type::Imaginary,         Type::IndepOf,
   Type::Indeterminate,     Type::Integer,
   Type::Intersection,      Type::Interval,
   Type::ListOf,            Type::ListProduct,
   Type::NegInt,            Type::NegRat,
   Type::Negative,          Type::NonNegInt,
   Type::NonNegRat,         Type::NonNegative,
   Type::NonZero,           Type::Numeric,
   Type::Odd,               Type::PolyExpr,
   Type::PolyOf,            Type::PosInt,
   Type::PosRat,            Type::Positive,
   Type::Predicate,         Type::Prime,
   Type::Product,           Type::Property,
   Type::RatExpr,           Type::Rational,
   Type::Real,              Type::Relation,
   Type::Residue,           Type::SequenceOf,
   Type::Series,            Type::Set,
   Type::SetOf,             Type::Singleton,
   Type::TableOf,           Type::TableOfEntry,
   Type::TableOfIndex,      Type::Union,
   Type::Unknown,           Type::Zero
```

Por exemplo, o especificador de tipo **Type::PostInt** representa o conjunto de integrais positivas $n > 0$, **type::NonNegInt** corresponde as integrais não negativas □ ≥ 0, **Type::Even** e **Type::Odd** representa as integrais even (par) e odd (ímpar), respectivamente. Esses tipos de especificadores são do tipo de domínio **Type**:

```
>> domtype(Type::Even)
   Type
```

Você pode usar tais tipos de especificadores para questionar a estrutura matemática dos objetos do MuPAD via **testtype**. No seguinte exemplo, nós extraímos todas as integrais pares de uma lista de integrais via **select** (Seção 4.6):

```
>> select([i $ i = 1..20], testtype, Type::Even)
   [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Você pode usar construtores tais como **Type::ListOf** ou **Type::SetOf** para executar checagem de tipo para listas ou conjuntos. Por exemplo, uma lista de integrais é emparelhada pelo tipo **Type::ListOf (DOM_INT)**, ou um conjunto de equações corresponde ao tipo de especificadores **Type::SetOf (Type::Equation( ))** (ou **Type::SetOf ("_equal"))**, um conjunto de integrais ímpares é emparelhado pelo tipo **Type::SetOf (Type::Odd**):

```
>> T := Type::ListOf(DOM_INT):
>> testtype([-1, 1], T), testtype({-1, 1}, T),
   testtype([-1, 1.0], T)
    TRUE, FALSE, FALSE
```

O construtor **Type::Union** gera tipos de especificadores correspondente a união de tipos mais simples. Por exemplo, o seguinte tipo de especificador

```
>> T := Type::Union(DOM_FLOAT, Type::NegInt, Type::Even):
```

Emparelha pontos flutuantes reais tão bem como integrais negativas tão bem como integrais pares:

```
>> testtype(-0.123, T), testtype(-3, T),
   testtype(2, T), testtype(3, T)
    TRUE, TRUE, TRUE, FALSE
```

Nós descrevemos uma aplicação do tipo de checagem para a implementação dos procedimentos do MuPAD na Seção 18.7.

Exercício 15.3: Como você pode computar a intersecção de um conjunto com o conjunto das integrais positivas?

Exercício 15.4: Use **?Type::ListOf** para consultar a página de ajuda para esse tipo de construtor. Construa o tipo especificador correspondente à lista de dois elementos tais que cada elemento é novamente uma lista três arbitrários elementos.

Capítulo 16 Loops

Loops são elementos importantes da linguagem de programação do MuPAD. O exemplo seguinte ilustra a forma mais simples de um loop **for**:

```
>> for i from 1 to 4 do
       x := i^2;
       print("The square of", i, "is", x)
   end_for:
```

A saída de tela é:

```
                "The square of", 1, "is", 1

                "The square of", 2, "is", 4

                "The square of", 3, "is", 9

                "The square of", 4, "is", 16
```

A variável de loop **i** automaticamente corre através dos valores 1, 2, 3, 4. Para cada valor de **i,** todos os comandos entre **do** e **end_for** são executados. Aí podem ser arbitrariamente muitos comandos, separados por semicolunas ou colunas. *O sistema não imprime os resultados computados em cada repetição na tela, até mesmo se você terminar os comandos por semicolunas*. Por esta razão, nós usamos o comando **print** para gerar uma saída no exemplo acima.
A variante seguinte conta para trás. Nós usamos as ferramentas da sessão 4.11 para fazer a saída parecer mais atraente:

```
>> for j from 4 downto 2 do
      print(Unquoted,
            "The square of ".expr2text(j)." is ".
            expr2text(j^2))
   end_for:
                  The square of 4 is 16

                   The square of 3 is 9

                   The square of 2 is 4
```

Você pode usar a palavra chave **step** para incrementar ou decrementar a variável loop em grandes passos:

```
>> for x from 3 to 8 step 2 do print(x, x^2) end_for:
                             3, 9

                            5, 25

                            7, 49
```

Note que no fim da repetição com $x = 7$ o valor de x é incrementado por 9. Isso excede o salto para acima de 8, e o loop termina.
Aqui está outra variante de loop **for**:

```
>> for i in [5, 27, y] do print(i, i^2) end_for:
                            5, 25

                           27, 729

                               2
                            y, y
```

A variável loop apenas corre através dos valores da lista [5, 27, y]. Como você pode ver, tal uma lista pode conter elementos simbólicos tais como a variável *y*.

Em um loop **for**, uma variável loop muda de acordo com regras fixas (tipicamente, isso é incrementado ou decrementado). O loop **repeat** é uma alternativa mais flexível, onde você pode modificar arbitrariamente muitas variáveis em cada passo. No seguinte exemplo, nós computamos as raízes das integrais i = 2, 2^2, 2^4, 2^8, ... até $i^2 > 100$ mantendo para a primeira vez:

```
>> x := 2:
>> repeat
       i := x; x := i^2; print(i, x)
   until x > 100 end_repeat:
                              2, 4

                              4, 16

                             16, 256
```

O sistema executa o comando entre **repeat** e **until** repetidamente. O loop termina quando a condição entre **until** e **end_repeat** mantém verdade. No exemplo acima, nós temos $i = 4$ e $x = 16$ no fim do segundo passo. Portanto o terceiro passo é executado, e depois de tudo nós temos $i = 16$, $x = 256$. Agora a condição de terminação $x > 100$ é satisfeita e o loop termina.

Outra variante de loop no MuPAD é o loop **while**:

```
>> x := 2:
>> while x <= 100 do
       i := x; x := i^2; print(i, x)
   end_while:
                              2, 4

                              4, 16

                             16, 256
```

No loop **repeat**, o sistema verifica a condição de término *após* cada repetição de loop. No loop **while**, a condição verificada é checada *antes* de cada repetição. Assim que como a condição avalia para **FALSE**, o sistema termina o loop **while**.

Você pode usar **break** para abortar um loop explicitamente. Tipicamente isso é feito dentro de uma construção **if** (Capítulo 17):

```
>> for i from 3 to 100 do
       print(i);
       if i^2 > 20 then break end_if
   end_for:
                              3

                              4

                              5
```

Depois de uma chamada para **next**, o sistema pula todos os comandos acima para **end_for**. Isso retorna imediatamente para o começo do loop e começa a próxima repetição com o próximo valor da variável loop:

```
>> for i from 2 to 5 do
       x := i;
       if i > 3 then next end_if;
       y := i;
       print(x, y)
   end_for:
                              2, 2

                              3, 3
```

Para $i > 3$, apenas a primeira tarefa **x: = i** é executada:

```
>> x, y
    5, 3
```

Nós revocamos que todo comando do MuPAD retorna um objeto. Para um loop, isso é o valor retornado do comando executado mais recentemente. Se você termina o comando loop com uma semicoluna (e não com uma coluna como em todos os exemplos acima), então MuPAD mostra esse valor:

```
>> delete x: for i from 1 to 3 do x.i := i^2 end_for
    9
```

Você pode processar esse valor futuramente. Em particular, você pode nomear ele para um identificador ou usar ele como o valor retornado de um procedimento do MuPAD (Capítulo 18):

```
>> factorial := proc(n)
     local result;
     begin
       result := 1;
       for i from 2 to n do
         result := result * i
       end_for
     end_proc:
```

O valor retornado do procedimento acima é o valor retornado do loop **for**, o qual na vez é o valor da última tarefa para **result**.

Internamente, loops são chamados de função de sistema. Por exemplo, MuPAD processa um loop **for** por avaliação da função **_for**:

```
>> _for(i, first_i, last_i, increment, command):
```

Isso é equivalente a

```
>> for i from first_i to last_i step increment do
       command
   end_for:
```

Capítulo 17 Ramificando: se – então –outro **e** caso

Instruções de ramificação são um elemento importante de toda a linguagem de programação. Dependendo no valor ou o sentido de variáveis, comandos diferentes são executados. A variante mais simples no MuPAD é a declaração **if**:

```
>> for i from 2 to 4 do
     if isprime(i)
        then print(expr2text(i)." is prime")
        else print(expr2text(i)." is not prime")
     end_if
   end_for:
                        "2 is prime"

                        "3 is prime"

                     "4 is not prime"
```

Aqui, o teste primalidade **isprime (i)** retorna se **TRUE** OU **FALSE**. Se o valor é **TRUE**, o sistema executa o comando entre **então** e **outro** (nesse caso, apenas o comando **print**). Se isso é **FALSE**, os comandos entre **else** e **end_if** são executados. A ramificação **else** é opcional:

```
>> for i from 2 to 4 do
     if isprime(i)
        then text := expr2text(i)." is prime";
             print(text)
     end_if
   end_for:
                        "2 is prime"

                        "3 is prime"
```

Aqui, a ramificação **then** cumpre dois comandos separados por uma semicoluna (ou, alternativamente, uma coluna). Você pode aninhar comandos, loops, e declarações ramificadas arbitrariamente:

```
>> primes := []: evenNumbers := []:
>> for i from 30 to 50 do
     if isprime(i)
        then primes := primes.[i]
        else if testtype(i,Type::Even)
                then evenNumbers := evenNumbers.[i]
             end_if
     end_if
   end_for:
```

Nesse exemplo, nós inspecionamos as integrais entre 30 e 50. Se nós encontramos um primo, então nós juntamos esse para a lista **primes**. Do contrário, nós usamos **testtype** para checar se **i** é igual (veja Seções 15.1 e 15.2). Nesse caso, nós juntamos **i** para a lista **evenNumbers**. Após a terminação, a lista **primes** contém todos os números primos entre 30 e 50, enquanto **evenNumbers** contém todas as integrais iguais nesse alcance:

```
>> primes, evenNumbers
   [31, 37, 41, 43, 47], [30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]
```

Você pode criar condições mais complexas para a declaração **if** usando os operadores Booleanos **and, or** e **not (**Seção 4.10). O seguinte loop **for** imprime todos os números primos gêmeos [i, i + 2] com $i \leq 100$. A condição alternativa **not** (**i >3**) rende o par adicional [2,4]:

```
>> for i from 2 to 100 do
     if (isprime(i) and isprime(i+2)) or not (i>3)
        then print([i,i+2])
     end_if
   end_for:
                          [2, 4]

                          [3, 5]

                          [5, 7]

                         [11, 13]

                            ...
```

Internamente, uma declaração **if** é apenas uma chamada de sistema de função **_if**:

```
>> _if(condition, command1, command2):
```

É equivalente a

```
>> if condition then command1 else command2 end_if:
```

Assim, os comandos

```
>> x := 1234567:
>> _if(isprime(x), print("prime"), print("not prime")):
```

Rende a saída:

```
                    "not prime"
```

O valor retornado de uma declaração **if** é, para qualquer procedimento do MuPAD, o resultado do último comando executado:

```
>> x := -2: if x > 0 then x else -x end_if
                              2
```

Por exemplo, você pode usar o operador seta -> **(**Seção 4.12) para implementar o valor absoluto dos números como seguem:

```
>> Abs := y -> (if y > 0 then y else -y end_if):
>> Abs(-2), Abs(-2/3), Abs(3.5)
                          2, 2/3, 3.5
```

Como você pode ver, você pode usar o comando **if** no MuPAD ambos no nível interativo e dentro de procedimentos. A aplicação típica é nos procedimentos programados do MuPAD, onde declarações **if** e loops controlam o fluxo do algoritmo. Um simples exemplo é a função acima **Abs**. Você encontra mais exemplos no Capítulo 18.

Se você tem várias construções **if ... else if ...** aninhados, você pode abreviar esses pelo uso da declaração **elif**:

```
>> if condition1 then
     statements1
   elif condition2 then
     statements2
   elif ...
   else
     statements
   end_if:
```

Isso é equivalente a seguinte declaração aninhada **if**:

```
>> if condition1 then
     statements1
   else if condition2 then
          statements2
        else if ...
             else
               statements
             end_if
        end_if
   end_if:
```

Uma aplicação típica é para checagem de tipo dentro de procedimentos (Capítulo 18). A seguinte versão de **Abs** computa o valor absoluto se a entrada é uma integral, um número racional, um número de pontos flutuantes real, ou um número complexo. Do contrário, ele acrescenta um erro:

```
>> Abs := proc(y) begin
     if (domtype(y) = DOM_INT) or (domtype(y) = DOM_RAT)
        or (domtype(y) = DOM_FLOAT) then
          if y > 0 then y else -y end_if;
     elif (domtype(y) = DOM_COMPLEX) then
          sqrt(Re(y)^2 + Im(y)^2);
     else "Invalid argument type" end_if:
   end_proc:
>> delete x: Abs(-3), Abs(5.0), Abs(1+2*I), Abs(x)
                  3, 5.0, sqrt(5), "Invalid argument type"
```

Em nosso exemplo, nós distinguimos vários casos de acordo com a avaliação de uma única expressão. Nós também podemos implementar isso usando uma declaração **case**, a qual é muitas vezes mais fácil de ler:

```
>> case domtype(y)
   of DOM_INT do
   of DOM_RAT do
   of DOM_FLOAT do
     if y > 0 then y else -y end_if;
     break;
   of DOM_COMPLEX do
     sqrt(Re(y)^2 + Im(y)^2);
     break;
   otherwise
     "Invalid argument type";
   end_case:
```

As palavras chave **case** e **end_case** indicam o começo e o fim da declaração, respectivamente. MuPAD avalia a expressão depois de **case.** Se o resultado parte uma das expressões entre **of** e **do**, o sistema executa todos os comandos da primeira partida **of** em até ela encontrar ou um **break** ou a palavra chave **end_case**.

Atenção: Note que, se não houver declaração **break** usada em uma ramificação, as seguintes ramificações são inseridas e executadas, também,

Isso está no mesmo estilo como a declaração **switch** na linguagem de programação C. Isso permite várias ramificações para compartilhar o mesmo código.

Se nenhuma dessas ramificações **of** aplicam e há uma ramificação **otherwise**, o código entre **otherwise** e **end_case** é executado. O valor retornado entre **othewrwise** e **end_case** é executado. O valor retornado de uma declaração **case** é o valor do último comando executado. Nós nos referimos a página de ajuda correspondente **?case** para uma descrição mais detalhada.

Como para loops e declarações **if**, há uma equivalente funcional para uma declaração **case**: a função do sistema **_case**. Internamente, MuPAD converte a seguinte declaração **case** para a seguinte forma equivalente:

```
>> _case(domtype(y),
         DOM_INT, NIL,
         DOM_RAT, NIL,
         DOM_FLOAT,
           (if y > 0 then y else -y end_if; break),
         DOM_COMPLEX, (sqrt(Re(y)^2 + Im(y)^2); break),
         "Invalid argument type"):
```

Exercício 17.1: Em declarações **if** ou condições de terminação de loops **while** e **repeat**, o sistema avalia condições compostas com operadores Booleanos um após o outro. A rotina de avaliação para prematuramente se ela pode decidir entre o resultado final ser **TRUE** ou **FALSE** (“preguiçosa avaliação”). Há problemas com as declarações seguintes? O que acontece quando as condições são avaliadas?

```
>> A := x/(x - 1) > 0: x := 1:
>> (if x <> 1 and A then right else wrong end_if),
   (if x = 1 or A then right else wrong end_if)
```

Capítulo 18 Procedimentos do MuPAD

MuPAD provê os construtores essenciais de uma linguagem de programação. O usuário pode implementar algoritmos complexos confortavelmente no MuPAD. Realmente, a maioria da inteligência matemática do MuPAD não é implementada em C ou C++ dentro do kernel, mas na linguagem de programação do MuPAD no nível da biblioteca. A s características da programação são mais extensivas que em outras linguagens tais como C, Pascal, ou Fortran, desde que MuPAD oferece construções mais gerais e mais flexíveis.

Nós já tivemos apresentado estruturas básicas como loops (Capítulo 16), instruções de ramificações (Capítulo 17), e funções “simples” (Seção 4.12).

Nesse capítulo, nós consideramos “programação” como escrita complexa de procedimentos do MuPAD. Em princípio o usuário não reconhece diferenças entre “funções simples” geradas via **->** (Seção 4.12) e “procedimentos mais complexos” são apresentados neste capítulo. Procedimentos, como funções, retornam valores. Apenas o modo de gerar tais objetos de procedimento via **proc-end_proc** é um pouco mais complicado. Procedimentos proveem funcionalidade adicional: há uma distinção entre variáveis locais e globais, voc~e pode usar arbitrariamente muitos comandos em um modo limpo e conveniente etc.

Tão logo como um procedimento é implementado e nomeado para um identificador, voc~e pode chamar ele na forma **procedureName (arguments)** como qualquer outra função do MuPAD. Depois de executar o algoritmo implementado, ele retorna um valor de saída.

Você pode definir e usar procedimentos do MuPAD dentro de uma sessão interativa, como qualquer outro objeto do MuPAD. Tipicamente, contudo, você quer usar esses procedimentos novamente em sessões futuras, em particular quando eles implementam algoritmos mais complexos. Então isso é útil para escrever o procedimento dentro de um arquivo de texto usando seu editor de texto favorito, e lê-lo dentro de uma sessão do MuPAD via **read** (Seção 13.2.1). Separado do nível de avaliação, o kernel do MuPAD processa os comandos no arquivo exatamente da mesma forma como se eles entraram interativamente. As versões de Windows do MuPAD Pro incluem um editor com sintaxe realçada para programas do MuPAD.

18.1 Definindo procedimentos

A seguinte função, a qual compara dois números e retorna seu máximo, é um exemplo de uma definição de procedimento via **proc-end_proc**:

```
>> Max := proc(a, b) /* comment: maximum of a and b */
          begin
            if a<b then return(b) else return(a) end_if
          end_proc:
```

O texto incluído entre /* e /* é um comentário e completamente ignorado pelo sistema. Isso é uma ferramenta útil para documentar o código fonte quando você escreve a definição de procedimento em um arquivo de texto.

A amostra de procedimento acima contém uma declaração **if** como o único comando. Procedimentos mais realísticos contêm muitos comandos (separados por colunas ou semicolunas). O comando **returnn** termina um procedimento e passa seu argumento como valor de saída para o sistema.

Um objeto do MuPAD gerado via **proc-end_proc** é do tipo de domínio **DOM_PROC**:

```
>> domtype(Max)
    DOM_PROC
```

Você pode decompor e manipular um procedimento como qualquer outro objeto do MuPAD. Em particular, você pode nomear isso para um identificador, como acima. A sintaxe da chamada de função é o mesmo que para outras funções do MuPAD:

```
>> Max(3/7, 0.4)
    3
    -
    7
```

As declarações entre **Begin** e **end_proc** podem ser comandos arbitrários do MuPAD. Em particular, você pode chamar funções do sistema ou outros procedimentos de dentro de um procedimento. Um procedimento pode se chamar até mesmo, do qual é útil para implementar algoritmos recursivos. O exemplo favorito é o fatorial n! = n.(n – 1)! Junto com a condição inicial 0!=1. A realização como um procedimento como um procedimento recursivo pode parecer como se segue:

```
>> factorial := proc(n) begin
     if n = 0 then
          return(1)
     else return(n*factorial(n - 1))
     end_if
   end_proc:
>> factorial(10)
    3628800
```

A variável ambiente **MAXDEPTH** determina o número máximo de chamadas de procedimentos aninhadas. Seu valor padrão é 500. Com esse valor, a função fatorial acima trabalha apenas para $n \leq 500$. Para altos valores, depois que os passos **MAXDEPTH**, você pode computar fatoriais para valores largos.

18.2 O valor retornado de um procedimento

Quando você chama um procedimento, o sistema executa seu corpo, por exemplo, a sequência de declarações entre **Begin** e **end_proc**. Todo procedimento retorna algum valor, cada explicitamente via **return** ou do contrário *o valor do último comando executado dentro do procedimento*. Assim, você pode implementar a função fatorial acima sem usar **return**

```
>> factorial := proc(n) begin
     if n = 0 then 1 else n*factorial(n - 1) end_if
   end_proc:
```

Se a declaração **if** retorna ou 1 ou n $(n – 1)!$. Desde que o fim do procedimento é alcançado diretamente após a declaração **if**, isso é o valor retornado da chamada **factorial (n).**

Tão logo como o sistema encontra uma declaração **return**, ele termina o procedimento:

```
>> factorial := proc(n) begin
     if n = 0 then return(1) end_if;
     n*factorial(n - 1)
   end_proc:
```

Para $n = 0$, MuPAD não executa a última declaração (a chamada recursiva de **n * factorial (n -1)**) após retornando 1. Para $n \neq 0$, o valor mais recentemente computado é **n * factorial (n – 1)**, o qual é então o valor retornado da chamada **factorial (n).**

Um procedimento pode retornar um objeto arbitrário do MuPAD, tal como uma expressão, uma sequência, um conjunto, uma lista, ou até mesmo um procedimento. Contudo, se o procedimento retornado usa variáveis locais do procedimento saído, você tem para declarar o depois com a opção **escape**. Do contrário, isso leva para uma mensagem de aviso do MuPAD ou para efeitos não desejados. O seguinte procedimento retorna uma função que usa o parâmetro **power** do procedimento exterior:

```
>> generatePowerFunction := proc(power)
          option escape;
          begin
            x -> (x^power)
          end_proc:
>> f := generatePowerFunction(2):
>> g := generatePowerFunction(5):
>> f(a), g(b)
    2  5
   a , b
```

18.3 Retornando chamadas de funções simbólicas

Muitas funções do sistema retornam “elas” como chamadas de função simbólica se elas não podem encontrar uma representação simples do resultado requisitado:

```
>> sin(x), max(a, b), int(exp(x^3), x)
```

$$\sin(x),\ \max(a,b),\ \int \mathrm{e}^{x^3}\,\mathrm{d}x$$

Você alcança o mesmo comportamento em seu próprio procedimento quando você encapsula o nome do procedimento em um **hold** após retornar. O **hold** (Seção 5.2) impede a função de chamar por si mesma recursivamente e terminando em uma recursão infinita. A função seguinte computa o valor absoluto para entradas numéricas (integrais, números racionais, números de pontos flutuantes reais, e números complexos). Para todos os outros gêneros de entradas, essas retornam a si mesmas simbolicamente:

```
>> Abs := proc(x) begin
     if testtype(x, Type::Numeric) then
        if domtype(x) = DOM_COMPLEX then
             return(sqrt(Re(x)^2 + Im(x)^2))
        else if x >= 0 then
                  return(x)
             else return(-x)
             end_if
        end_if
     end_if;
     hold(Abs)(x)
   end_proc:
>> Abs(-1), Abs(-2/3), Abs(1.234), Abs(2 + I/3),
   Abs(x + 1)
```

$$1,\ \frac{2}{3},\ 1.234,\ \frac{\sqrt{37}}{3},\ \mathrm{Abs}(x+1)$$

Um modo mais elegante é usar o objeto do MuPAD **procname**, o qual retorna o nome da chamada de procedimento:

```
>> Abs := proc(x) begin
     if testtype(x, Type::Numeric) then
        if domtype(x) = DOM_COMPLEX then
             return(sqrt(Re(x)^2 + Im(x)^2))
        else if x >= 0 then
                  return(x)
             else return(-x)
             end_if
        end_if
     end_if;
     procname(args())
   end_proc:
>> Abs(-1), Abs(-2/3), Abs(1.234), Abs(2 + I/3),
   Abs(x + 1)
```

$$1,\ \frac{2}{3},\ 1.234,\ \frac{\sqrt{37}}{3},\ \mathrm{Abs}(x+1)$$

Aqui, nós usamos a expressão **args** (), a qual retorna a sequência de argumentos para o procedimento (Seção 18.8).

18.4 Variáveis locais e globais

Você pode usar identificadores arbitrários em procedimentos. Eles também são chamados *variáveis locais*:

```
>> a := b: f := proc() begin a := 1 + a^2 end_proc:
>> f(); f(); f()
```

$$b^2+1$$

$$\left(b^2+1\right)^2+1$$

$$\left(\left(b^2+1\right)^2+1\right)^2+1$$

O procedimento **f** modifica o valor de **a**, o qual foi fixo fora do procedimento. Quando o procedimento termina, **a** tem um novo valor, o qual é novamente mudado para futuras chamadas à **f**.

A palavra chave **local** declara identificadores como *variáveis locais* que são apenas validas dentro do procedimento:

```
>> a := b: f := proc() local a; begin a := 2 end_proc:
>> f(): a
```

$$b$$

A despeito de nomes iguais, a declaração **a:=2** da variável local não afeta o valor do identificador global **a** que foi definido do lado de fora do procedimento. Você pode declarar um número arbitrário de variáveis locais especificando uma sequência de identificadores depois de **local**:

```
>> f := proc(x, y, z)
       local A, B, C;
       begin
          A:= 1; B:= 2; C:= 3; A*B*C*(x + y + z)
       end_proc:
>> f(A, B, C)
   6 A + 6 B + 6 C
```

Variáveis locais de um procedimento tem um tipo de domínio especial **DOM_VAR**. Eles não são confundidos com variáveis locais as quais são identificadores do tipo **DOM_IDENT.** Note que variáveis locais também declararam pelo mesmo nome no código fonte de diferentes funções sem ter referencia para uma outra. Também, quando chama uma função várias vezes, uma variável local refere-se para um valor diferente em cada chamada:

```
>> f := proc(x) local a, b;
   begin
     a := x;
     if x > 0 then
          b := f(x - 1);
     else b := 1;
     end_if;
     print(a, x);
     b + a;
   end:
   f(2)
                          0, 0

                          1, 1

                          2, 2

    4
```

Nós recomendamos levar em conta a seguinte regra de polegar:

Usando variáveis locais é geralmente considerado um estilo de programação ruim. Use variáveis locais sempre que possível

As razões para esse princípio é que procedimentos implementam funções matemáticas, as quais devem retornar um único valor de saída para um conjunto dado de valores de entrada. Se você usar variáveis locais então, dependendo nos seus valores, o mesmo procedimento de chamada pode levar a diferentes resultados:

```
>> a := 1: f := proc(b) begin a := a + 1; a + b end_proc:
>> f(1), f(1), f(1)
   3, 4, 5
```

Além disso, uma chamada procedimento pode mudar o ambiente chamando em um modo sutil de redefinição de variáveis locais (“efeito do lado”). Em programas mais complexos, isso pode levar a efeitos indesejados que são difíceis de depurar.

Uma diferença importante entre variáveis globais e locais é que uma variável local não inicializada é considerada como um *símbolo*, o qual o valor é o seu próprio nome, enquanto o valor de uma variável local não inicializada é **NIL**. Usando uma variável local sem inicialização leva a mensagem de aviso no MuPAD e deve ser evitada:

```
>> IAmGlobal + 1
    IAmGlobal + 1
>> f := proc()
     local IAmLocal;
     begin
       IAmLocal + 1
     end_proc:
>> f()
   Warning: Uninitialized variable 'IAmLocal' used;
   during evaluation of 'f'
   Error: Illegal operand [_plus];
   during evaluation of 'f'
```

A razão do erro é que MuPAD não pode adicionar o valor **NIL** da variável local para o número 1.

Nós agora representamos um exemplo realístico de um procedimento de sentido completo. Se nós usamos ordens do tipo de domínio **DOM_ARRAY** para representar matrizes, nós estamos em frente com os problemas que não há modo direto no MuPAD para executar multiplicação de matriz com tais ordens. O procedimento seguinte resolve esse problema: você pode computar o produto da matriz $C = A . B$ com os comandos
C:=MatrixProduct (A, B). Nós queremos o procedimento para trabalhar de dimensões arbitrárias das matrizes A e B, contanto o resultado é definido matematicamente. Se *A* é uma matriz *m x n*, então B pode ser uma matriz *n x r*, onde *m*, *n*, *r* são integrais positivas arbitrárias. O resultado é a matriz *m x r C* com as entradas

O procedimento da multiplicação abaixo extrai automaticamente os parâmetros dimensionais *m*, *n*, *r* dos argumentos, isto é do 0º

$$C_{ij} = \sum_{k=1}^{n} A_{ik}\, B_{kj}\ , \quad i = 1,\ldots,m, \quad j = 1,\ldots,r.$$

operando das ordens de entrada (Seção 4.9). Se B é uma matriz *q x r* com $q \neq n$, a multiplicação não é definida matematicamente. Nesse caso, o procedimento termina com uma mensagem de erro. Para este propósito, nós empregamos a função **error**, a qual aborta o procedimento de chamada e escreve a sequência passada como argumento na tela. Nós armazenamos o componente do resultado pelo componente na variável local C. Nós inicializamos essa variável como uma ordem de dimensão *m* x *r*, assim que o resultado de nosso procedimento é do tipo de dado desejado **DOM_ARRAY**. Nós podemos

```
>> MatrixProduct := /* multiplication C=AB of an m x n */
   proc(A, B)       /* matrix A by an n x r Matrix B   */
   local m, n, r, i, j, k, C; /* with arrays A, B of   */
   begin                      /* domain type DOM_ARRAY */
       m := op(A, [0, 2, 2]);
       n := op(A, [0, 3, 2]);
       if n <> op(B, [0, 2, 2]) then
         error("incompatible matrix dimensions")
       end_if;
       r := op(B, [0, 3, 2]);
       C := array(1..m, 1..r);       /* initialization */
       for i from 1 to m do
         for j from 1 to r do
           C[i, j] := _plus(A[i, k]*B[k, j] $ k = 1..n)
         end_for
       end_for;
       C
   end_proc:
```

implementar a soma sobre *k* na computação de C_{ij} como um loop da forma **para k de 1 para n ..** Ao invés, nós usamos a função do sistema **_plus** o qual retorna a soma de seus argumentos. Nós geralmente recomendamos usar essas funções do sistema, se possível, desde que elas trabalham muito eficientemente. O valor retonado de **MatrixProduct** é a expressão final C:

Uma remarcação geral sobre estilo de programação no MuPAD: Você deve sempre executar argumento checando nos significados para uso interativo. Se você implementa um procedimento, você normalmente conhece tipos de entradas que são válidas (tais como **DOM_ARRAY** no exemplo acima). Se alguém passa parâmetros de tipo errado por engano, isso normalmente leva para sistemas de chamadas de funções com argumentos inválidos, e seu procedimento aborta com uma mensagem de erro originando de uma um sistema de função. No exemplo acima, a função **op** retorna o valor **FAIL** quando acessa o 0º operando de **A** e **B** e um deles não é do tipo **DOM_ARRAY.** Então esse valor é nomeado para **m, n** ou **r**, e o seguinte loop **for** aborta com uma mensagem de erro, desde que **FAIL** não é permitido como um valor de ponto final do loop.

Em uma situação parecida, é muitas vezes difícil de localizar a fonte do erro. Contudo, um cenário até pior pode acontecer: se o procedimento não aborta, o resultado é provável para ser errado! Assim, checagem de tipo ajuda a evitar erros.

No exemplo acima, nós podemos adicionar um tipo de conferir da forma

```
>> if domtype(A) <> DOM_ARRAY or domtype(B) <> DOM_ARRAY
     then error("arguments must be of type DOM_ARRAY")
   end_if
```

Para o corpo do procedimento. Na Seção 18.7, nós discutimos um tipo mais simples de checagem de conceito.

18.5 Subprocedimentos

Muitas vezes tarefas ocorrem frequentemente dentro de um procedimento e você quer implementar eles novamente na forma de um procedimento. Essa estrutura e simplifica o código do programa. Em muitos casos, este tal procedimento é usado apenas de dentro de um único procedimento. Então isso é razoável para definir esse procedimento localmente como um subprocedimento apenas na extensão de procedimento chamado. No MuPAD voc~e pode usar variáveis locais para implementar subprocedimentos. Se você quer fazer

```
g := proc() begin ... end_proc:
```

um procedimento local de

```
f := proc() begin ... end_proc:
```

define **f** como se segue:

```
>> f := proc()
   local g;
   begin
     g := proc() begin ... end_proc;   /* subprocedure */

     /* main part of f, which calls g(..): */
     ...
   end_proc:
```

Agora, **g** é um procedimento *local* de **f** e você pode usar ele apenas de dentro de **f.**

Nós demos um exemplo. Você pode implementar matriz multiplicação pelos sentidos de coluna×fileira satisfatórias de multiplicações:

$$\begin{pmatrix} 2 & 1 \\ 5 & 3 \end{pmatrix} \cdot \begin{pmatrix} 4 & 6 \\ 2 & 3 \end{pmatrix} = \begin{pmatrix} (2,1) \cdot \begin{pmatrix} 4 \\ 2 \end{pmatrix} & (2,1) \cdot \begin{pmatrix} 6 \\ 3 \end{pmatrix} \\ (5,3) \cdot \begin{pmatrix} 4 \\ 2 \end{pmatrix} & (5,3) \cdot \begin{pmatrix} 6 \\ 3 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 10 & 15 \\ 26 & 39 \end{pmatrix}.$$

Mais geralmente, se nós partição a entrada de matrizes por fileiras a_i e colunas b_j, respectivamente, então

$$\begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \cdot (b_1, \ldots, b_n) = \begin{pmatrix} a_1 \cdot b_1 & \ldots & a_1 \cdot b_n \\ \vdots & \ddots & \vdots \\ a_m \cdot b_1 & \ldots & a_m \cdot b_n \end{pmatrix},$$

Com o produto de entrada

$$a_i \cdot b_j = \sum_r (a_i)_r \, (b_j)_r \, .$$

Nós agora escrevemos um procedimento **MatrixMult** que espera ordens de entrada *A* e *B* da forma **array (1..m, 1..k)** e **array (1..k, 1..n)**, e retorna a matriz $m \times n$ produto $A \, . \, B$. Uma chamada de subprocedimento **RowTimesColumn** com argumentos *i, j* extraí a *i*º fileira e a *i*ª coluna das matrizes de entrada *A* e *B*, respectivamente, e computa o produto de entrada da fileira e da coluna. Os subprocedimentos usam as ordens **A, B** tão bem como os parâmetros de dimensões localmente declaradas **m, n, e k** como variáveis “globais”:

```
>> MatrixMult := proc(A, B)
   local m, n, k, K,     /* local variables */
         RowTimesColumn; /* local subprocedure */
   begin
     /* subprocedure */
     RowTimesColumn := proc(i, j)
     local row, column, r;
     begin
       /* ith row of A: */
       row := array(1..k, [A[i,r] $ r=1..k]);
       /* jth column of B: */
       column := array(1..k, [B[r,j] $ r=1..k]);
       /* row times column */
       _plus(row[r]*column[r] $ r=1..k)
     end_proc;

     /* main part of the procedure MatrixMult: */
     m := op(A, [0, 2, 2]); /* number of rows of A */
     k := op(A, [0, 3, 2]); /* number of columns of A */
     K := op(B, [0, 2, 2]); /* number of rows of B */
     n := op(B, [0, 3, 2]); /* number of columns of B */

     if k <> K then
        error("# of columns of A <> # of rows of B")
     end_if;

     /* matrix A*B: */
     array(1..m, 1..n,
           [[RowTimesColumn(i, j) $ j=1..n] $ i=1..m])
   end_proc:
```

O seguinte exemplo retorna o resultado desejado:

```
>> A := array(1..2, 1..2, [[2, 1], [5, 3]]):
>> B := array(1..2, 1..2, [[4, 6], [2, 3]]):
>> MatrixMult(A, B)
```

$$\begin{pmatrix} 10 & 15 \\ 26 & 39 \end{pmatrix}$$

18.6 Extensão de variáveis

A linguagem de programação do MuPAD implementa *extensão léxica*. Isso significa essencialmente que a extensão de um procedimento de variáveis locais e parâmetros podem já ser determinados quando o procedimento é definido. Nós começamos com um simples exemplo para explicar esse conceito.

```
>> p := proc() begin x end_proc:
>> x := 3: p(); x := 4: p()
    3

    4
>> q := proc() local x; begin x := 5; p(); end_proc:
>> q()
    4
```

Primeiro, um procedimento **p** sem argumentos é definido. Ele usa uma variável **x**, a qual não é declarada como uma variável local de **p**. Assim, a chamada **p( )** retorna o valor da variável global **x**, como se mostra nas duas chamadas subsequentes. No

procedimento **q**, contudo, a variável **x** é declarada local, e o valor 5 é nomeado para ela. A variável local **x** não é visível de dentro do procedimento **q**, apenas a variável local **x** não é visível de dentro do procedimento **q**, apenas a variável local **x** pode ser acessada. Não obstante, a chamada **q ( )** retorna o valor da variável *global* **x**, e *não* o valor corrente da variável local **x** dentro de
p. Nós podemos, por exemplo, definir um subprocedimento dentro de **q** para alcançar o comportamento depois:

```
>> x := 4:
>> q := proc()
     local x, p;
     begin
       x := 5;
       p := proc() begin x; end_proc;
       x := 6;
       p()
     end_proc:
>> q(), p()
    6, 4
```

O procedimento global previamente definido **p** não é acessado de dentro de **q** porque uma variável local **p** é declarada. A chamada do procedimento local **p** de dentro de **q** agora realmente retorna o valor corrente da variável local **x**, como mostrado pela chamada **q( )**. O último comando **p( )**, contudo, executa o procedimento **p** definido' antes, o qual ainda retorna o valor corrente 4 da variável local **x**.

Aqui está outro exemplo:

```
>> p := proc(x) begin 2 * cos(x) + 1; end_proc:
>> q := proc(y)
     local cos;
     begin
       cos := proc(z) begin z + 1; end_proc;
       p(y) * cos(y)
     end_proc:
>> p(PI), q(PI)
    -1, -π - 1
```

O procedimento **p** usa a função do MuPAD globalmente definida coseno. Dentro de **q**, nós definimos um subprocedimento local **cos**. Chamando **q**, a chamada interna **cos (y)** refere-se à função local **cos**. Até mesmo quando chamada de dentro de **q**, o procedimento **p** ainda usa a função global do MuPAD coseno.

Quando usando a opção **escape**, uma variável local pode ser acessada até mesmo quando ela escapou da extensão do procedimento que esse local foi para. No seguinte exemplo, o "contador" retornado por **f** refere-se a variável local **x** de **f**. Os procedimentos **counter1, counter2** etc. não são locais para **f** mas mais ainda usa a referencia para **x** para sincronizar chamadas subsequentes de cada contador. Note que ambos os contadores referem-se para *diferentes* instancias independentes de **x**, assim que eles contam independentemente:

```
>> f := proc() local x;
       option escape;
       begin
         x := 1;
         // The following procedure
         // is the return value of f:
         proc() begin x := x + 1 end;
       end:
>> counter1 := f(): counter2 := f():
>> counter1(), counter1(), counter1();
   counter2(), counter2();
   counter1(), counter2(), counter1();
    2, 3, 4

    2, 3

    5, 4, 6
```

18.7 Tipo declaração

MuPAD provê tipo de checagem fácil de usar para argumentos de procedimentos. Por exemplo, você pode restringir os argumentos de **MatrixProduct**,m um procedimento da Seção 18.4, para o tipo de domínio **DOM_ARRAY** como segue-se:

```
>> MatrixProduct := proc(A: DOM_ARRAY, B: DOM_ARRAY)
                    local m, n, r, i, j, k, C;
                    begin ...
```

Se você declarar o tipo de parâmetro de um procedimento na forma usada acima, **argument: typeSpecifier**, uma chamada do procedimento com parâmetros de um tipo incompatível leva à uma mensagem de erro. No exemplo acima, nós usamos o tipo de domínio **DOM_ARRAY** como um especificador de tipo.

Nós discutimos o conceito de tipo do MuPAD no Capítulo 15. O tipo da biblioteca **Type** oferece **Type::NonNegInt** para representar o conjunto de integrais não negativas. Se nós usamos isso na seguinte variante da função fatorial

```
>> factorial := proc(n: Type::NonNegInt) begin
     if n = 0 then
          return(1)
     else n*factorial(n - 1)
     end_if
   end_proc:
```

Então somente integrais não negativas são permitidas para o argumento **n**:

```
>> factorial(4)
    24
>> factorial(4.0)
   Error: Wrong type of 1. argument (type 'Type::NonNeg\
   Int' expected,
          got argument '4.0');
   during evaluation of 'factorial'

>> factorial(-4)
   Error: Wrong type of argument 'n' (type 'Type::NonNeg\
   Int' expected,
          got argument '-4');
   during evaluation of 'factorial'
```

18.8 Procedimentos com um número variável de argumentos

A função do sistema **Max** computa o máximo de seus argumentos. Você pode chamar isso com tantos argumentos arbitrariamente:

```
>> max(1), max(3/7, 9/20), max(-1, 3, 0, 7, 3/2, 7.5)
```

$$1,\ \frac{9}{20},\ 7.5$$

Você pode implementar esse comportamento em seus próprios procedimentos também. A função **args** retorna os argumentos passados para uma chamada de procediemnto:

args (0): o número de argumentos,
args (i): o i^{o} argumento, $1 \le i \le$ **args (0)**,
args (i..j): a sequência de argumentos de i até j,
$1 \le i \le j \le$ **args (0)**,

args ( ): a sequência **args (1), args (2)...** de todos os argumentos.

A seguinte função simula o comportamento da função do sistema **max**:

```
>> Max := proc() local m, i; begin
     m := args(1);
     for i from 2 to args(0) do
         if m < args(i) then m := args(i) end_if
     end_for:
     m
   end_proc:
>> Max(1), Max(3/7, 9/20), Max(-1, 3, 0, 7, 3/2, 7.5)
```

$$1,\ \frac{9}{20},\ 7.5$$

Aqui, nós inicializamos **m** com o primeiro argumento. Então, nós testamos para cada dos argumentos restantes se este é maior que **m**, e se assim, substitui **m** pelo argumento correspondente. Assim, **m** contém o máximo no fim do loop. Note que se você chama **Max** com apenas um argumento (assim que **args (0) = 1**), então o loop **para i de 2 para 1 faz ...** não é executado para tudo.

Você pode usar ambos os parâmetros formais e acessar via **args** em um procedimento:

```
>> f := proc(x, y) begin
        if args(0) = 3 then
             x^2 + y^3 + args(3)^4
        else x^2 + y^3
        end_if
      end_proc:
>> f(a, b), f(a, b, c)
```

$$a^2 + b^3,\ a^2 + b^3 + c^4$$

O seguinte exemplo é uma função trivial retornando por si mesma simbolicamente para qualquer número de argumentos que são atualmente passados:

```
>> f := proc() begin procname(args()) end_proc:
>> f(1), f(1, 2), f(1, 2, 3), f(a1, b2, c3, d4)
```

$$f(1),\ f(1,2),\ f(1,2,3),\ f(\mathrm{a1},\mathrm{b2},\mathrm{c3},\mathrm{d4})$$

18.9 Opções: a tabela de lembrança

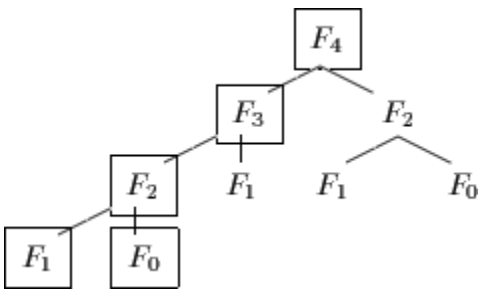
Quando declara procedimentos do MuPAD, você pode especificar *opções* que afetam a execução de uma chamada de procedimento. Por trás da opção **hold** a opção **remember** pode ser de interesse do usuário. Nesta seção, nós levamos uma olhada próxima nesta opção e demonstramos seus efeitos com um exemplo. A sequência de números de Fibonacci é definida pela recursão

$$F_n = F_{n-1} + F_{n-2}, \qquad F_0 = 0, F_1 = 1$$

Isso é fácil de traduzir isso dentro de um procedimento do MuPAD:

```
>> F := proc(n) begin
        if n < 2 then n else F(n - 1) + F(n - 2) end_if
      end_proc:
>> F(i) $ i = 0..10
   0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

Esse modo de computar F_n é altamente eficiente para valores largos de *n*. Para ver como, deixe-nos traçar as chamadas recursivas de **F** quando computando $\mathbf{F_4}$. Você pode considerar isso como uma estrutura de árvore: **F (4)** chama **F (3)** e **F (2)**, **F (3)** chama **F (2)** e **F (1)** etc.:



Um pode mostrar que a chamada **F(n)** leva para por volta $1.45..^{\cdot}(1.618..)^n$ chamadas de **F** para largo *n*. Esses "preços" crescem dramaticamente rápido por acrescentar valores de *n*:

```
>> time(F(10)), time(F(15)), time(F(20)), time(F(25))
   80, 910, 10290, 113600
```

Quando recorda que a função **time** retorna o tempo em milissegundos usados para avaliar esse argumento.

Nós vemos que muitas chamadas (tais como, por exemplo, **F (1))** são executadas diversas vezes. Para avaliar **F (4)**, é eficiente executar apenas as chamadas de funções encaixotadas **F(0) ,..., F (4)** na figura acima e para armazenar esses valores. Todas as outras computações de **F (0), F (1), F (2)** são redundantes desde que seus resultados já são conhecidos. Isso é precisamente o que MuPAD faz quando você declara **F** com a **option remember**:

```
>> F := proc(n)
       /* local x, y; (declare local variables here) */
       option remember;
       begin
         if n < 2 then n else F(n - 1) + F(n - 2) end_if
       end_proc:
```

O sistema cria internamente uma tabela lembrança para o procedimento **F**, o qual inicialmente está vazio. Em cada chamada para **F**, MuPAD checa se ali há uma entrada para o argumento sequencia nessa tabela. Se este for o caso, o procedimento não é executado em tudo, e o resultado é levado da tabela. Se os argumentos correntes não aparecem na tabela, o sistema executa o corpo procedimento como usual e retorna seu resultado. Então isso junta o argumento sequência e o valor retornado para a tabela lembrança. Isso assegura que um procedimento não é desnecessariamente executado duas vezes com os mesmos argumentos.

No exemplo de Fibonacci, a chamada **F (n)** agora leva para apenas chamadas $n + 1$ para computar **F(0),. . ., F(n)**. Em adição, o sistema pesquisa a tabela lembrança $n - 2$ vezes. Contudo, isso acontece muito rapidamente. Nesse exemplo, o benefício na velocidade de usando **option remember** é bastante dramático:

```
>> time(F(10)), time(F(15)), time(F(20)), time(F(25)),
   time(F(500))
    0, 10, 0, 10, 390
```

Os tempos correndo reais são tão pequenos que o sistema não pode medir eles exatamente. Isso explica os (arredondados) tempos de 0 milissegundos para **F(10)** e **F(20)**.

Usando **option remember** em um procedimento é promissor sempre que um procedimento é chamado frequentemente com os mesmos argumentos.

Certamente, você pode simular esse comportamento dos números de Fibonacci diretamente computando F_n iterativamente ao invés de recursivamente e armazenando valores já computados em uma tabela:

```
>> F := proc(n) local i, F; begin
     F[0] := 0: F[1] := 1:
     for i from 2 to n do
         F[i] := F[i - 1] + F[i - 2]
     end_for
   end_proc:
>> time(F(10)), time(F(15)), time(F(20)), time(F(25)),
   time(F(500))
    10, 0, 10, 0, 400
```

A função **numlib::fibonacci** da biblioteca de teoria dos números é ainda mais rápida para largos argumentos.

Atenção: O mecanismo de lembrança reconhece apenas entradas previamente processadas, mas não considera os valores de variáveis globais possivelmente. Quando os valores dessas variáveis globais mudam, então os lembrados retornam valores que são usualmente errados. Em particular, esse é o caso para ambientes globais variáveis tais como **DIGITS**:

```
>> floatexp := proc(x) option remember;
              begin float(exp(x)) end_proc:
>> DIGITS := 20: floatexp(1);
    2.7182818284590452354
>> DIGITS := 40: floatexp(1); float(exp(1))
    2.7182818284590452353602874713449239278 42

    2.7182818284590452353602874713526624977 57
```

Aqui, as saídas do sistema o valor lembrado de **floatexp (1)** com alta precisão depois trocando de 20 para 40 **DIGITS**. Não obstante, isso é ainda o valor computado com **DIGITS = 20**; a saída apenas mostra todos os dígitos que foram usados *internamente* nesta computação. O último dos três números é o verdadeiro valor de **exp (1)** computado com 40 dígitos. Isso difere de um valor lembrado erradamente no 30º dígito decimal.

Você pode adicionar explicitamente novas entradas para a tabela lembrança de um procedimento. No seguinte exemplo *f* é a função $x \mapsto \sin(x)/x$, a qual tem uma singularidade removível em $x = 0$. O limite é $\lim(x \to 0)\sin(x)/x = 1$:

```
>> f := proc(x) begin sin(x)/x end_proc: f(0)
   Error: Division by zero;
   during evaluation of 'f'
```

Você pode facilmente adicionar o valor de $x = 0$:

```
>> f(0) := 1: f(0)
    1
```

A tarefa **f (0):=1** cria uma tabela lembrança para **f**, assim que uma chamada depois de **f (0)** não tenta avaliar o valor de sin(x)/x para x=0. Agora você pode usar **f** sem correr dentro de perigo em $x = 0$ (por exemplo, você pode plotar ele).

Atenção: A chamada:

```
>> delete f: f(x) := x^2:
```

Não gera a função $f : x \mapsto x2$, mas um tanto cria uma tabela lembrança para o identificador **f** com a entrada **x^2** *apenas para o identificador simbólico* **x**. Qualquer outra chamada para **f** retorna uma chamada de função simbólica:

```
>> f(x), f(y), f(1)
    x^2, f(y), f(1)
```

18.10 Parâmetros de entrada

Os argumentos declarados formais de um procedimento podem ser usados como variáveis locais adicionais:

```
>> f := proc(a, b) begin a := 1; a + b end_proc:
>> a := 2: f(a, 1): a
    2
```

Modificando **a** adentro desse procedimento não afeta o identificador **a** que é usado por fora desse procedimento. Você deve ser cauteloso quando você acessa os argumentos chamada via **args** (Seção 18.8) em um procedimento depois de mudar alguns parâmetros de entrada. Tarefa para um parâmetro formal muda o valor retornado de **args**:

```
>> f := proc(a) begin a := 1; a, args(1) end_proc: f(2)
    1, 1
```

Em princípio, objetos arbitrários do MuPAD podem ser entradas de parâmetros de um procedimento. Desse modo você pode usar conjuntos, listas, expressões, ou até mesmp procedimentos e funções:

```
>> p := proc(f) begin
         [f(1), f(2), f(3)]
       end_proc:
>> p(g)
    [g(1), g(2), g(3)]
>> p(proc(x) begin x^2 end_proc)
    [1, 4, 9]
>> p(x -> x^3)
    [1, 8, 27]
```

Em geral, procedimentos definidos do usuário avaliam seus argumentos ("chamados pelo valor"): quando você chama **f (x)**, o procedimento **f** conhece apenas o valor do identificador **x**. Se você declara um procedimento com a **option hold**, então o esquema "chamado pelo nome" é usado ao invés disso: a expressão ou o nome dos identificadores passaram como o atual argumento é visto pelo procedimento. Nesse caso, você pode usar **context** para obter o valor da expressão:

```
>> f := proc(x) option hold;
       begin x = context(x) end_proc:
>> x := 2:
>> f(x), f(sin(x)), f(sin(PI))
```

$$x = 2,\ \sin(x) = \sin(2),\ \sin(\pi) = 0$$

18.11 Avaliando dentro de procedimentos

No Capítulo 5, nós discutimos a estratégia de avaliação do MuPAD: avaliação completa em um nível interativo. Consequentemente todos os identificadores são substituídos por seus valores *recursivamente* até apenas identificadores simbólicos sem um valor restante (ou a avaliação profunda dada pela variável ambiente **LEVEL** é alcançada):

```
>> delete a, b, c: x := a + b: a := b + 1: b := c:
>> x
```

$$2c + 1$$

Em contraste, dentro de procedimentos, o sistema executa avaliação não completamente mas apenas com avaliação de profundidade 1. Isso é similar para substituindo internamente cada identificador por **level (identifier, 1)**: todo identificador é substituído por seu valor, mas *não recursivamente*. Nós lembramos da Seção 5.1 a distinção entre o *valor* de um identificador (a avaliação no tempo da tarefa) e sua *avaliação* (o "valor corrente", onde identificadores simbólicos que foram nomeados um valor no entretanto são substituídos por seus valores também). No modo interativo, chamando um objeto rende sua completa avaliação, enquanto em procedimentos apenas o valor do objeto é retornado. Isso explica a diferença entre o resultado interativo acima e o seguinte resultado:

```
>> f := proc() begin
        delete a, b, c:
        x := a + b: a := b + 1: b := c:
        x
      end_proc:
>> f()
```

$$a + b$$

A razão por que dois comportamentos diferentes são implementados é que a estratégia da avaliação incompleta faz a avaliação de procedimentos mais rápida e acrescenta a eficiência dos procedimentos do MuPAD consideravelmente. Para um iniciante na programação do MuPAD, esse conceito de avaliação tem suas armadilhas. Contudo, depois de alguma prática você adquire um estilo de programação apropriado assim que você pode trabalhar com a avaliação restrita em profundidade sem problemas.

Atenção: Se você não trabalha interativamente com MuPAD mas usa um editor para escrever seus comandos do MuPAD dentro de um arquivo de texto do qual é lido dentro de uma sessão do MuPAD via **read** (Seção 13.2.1), esses comandos são executados dentro de um procedimento (nomeado, **read**). Consequentemente, a profundidade de avaliação é 1. Você pode usar a função do sistema **level** (Seção 5.2) para controlar a profundidade de avaliação e para forçar a completa avaliação se necessário:

```
>> f := proc() begin
        delete a, b, c:
        x := a + b: a := b + 1: b := c:
        level(x)
      end_proc:
>> f()
```

$$2c + 1$$

Atenção: level não avalia variáveis locais.

Além disso, você não pode usar variáveis locais como identificadores simbólicos; você deve inicializar eles antes de usar. O seguinte procedimento, no qual um identificador simbólico **x** é passado para uma função integrável, é inválido:

```
>> f := proc(n) local x; begin int(exp(x^n), x) end_proc:
```

Você pode passar o nome da variável de integração como argumento adicional para o procedimento. Desse modo, as seguintes variantes são válidas:

```
>> f := proc(n, x) begin int(exp(x^n), x) end_proc:
>> f := proc(n, x) local y; begin
       y := x; int(exp(y^n), y) end_proc:
```

Se você precisa de identificadores simbólicos para resultados intermediários, você pode gerar um identificador sem um valor via **genident ( )** (Seçaõ 4.3) e nomeie a uma variável local.

18.12 Ambientes de função

MuPAD provê uma variedade de ferramentas para controle de construção em funções matemáticas padrões tais como **sin**, **cos**, **exp**. Essas ferramentas implementam as propriedades

matemáticas dessas funções. Exemplos típicos são a rotina de conversão **float**, a função de diferenciação **diff**, ou a função **expand**, a qual você usa para manipular expressões:

```
>> float(sin(1)), diff(sin(x), x, x, x),
   expand(sin(x + 1))
```

$$0.8414709848,\ -\cos(x),\ \cos(1)\sin(x) + \sin(1)\cos(x)$$

Nesse sentido, o conhecimento matemático sobre as funções padrão é distribuído sobre vários sistemas de funções: a função **float** tem de conhecer como computar aproximações numéricas de função seno, **diff** deve conhecer sua derivada, e **expand** tem de conhecer os teoremas de adição de funções trigonométricas.

Você pode inventar arbitrariamente novas funções como nomes simbólicos ou implementar eles em procedimentos. Como você pode passar o conhecimento sobre o sentido matemático e as regras de manipulação para as funções novas para os outros sistemas de funções? Por exemplo, como você pode dizer a rotina diferenciação **diff** que a derivada de sua recente função criada está? Se a função é composta de funções padrão conhecidas para MuPAD tais como, por exemplo, f: $x \mapsto x \sin(x)$, então isso não é problema. A chamada

```
>> f := x -> (x*sin(x)): diff(f(x), x)
```

$$\sin(x) + x\cos(x)$$

Imediatamente rende a resposta desejada. Contudo, frequentemente há situações, onde a função recentemente implementada não pode ser composta de objetos padrão. Assim, nosso objetivo é dar as regras de manipulação (aproximações de pontos flutuantes, diferenciação etc.) para chamadas *simbólicas* de função para as funções do MuPAD **float, diff** etc. Esse é o atual desafio quando você "implementa uma nova função matemática no sentido do símbolo para as ferramentas padrão do MuPAD. Realmente, essa é uma tarefa necessária: por exemplo, se você quer diferenciar uma expressão mais complexa contendo ambas a nova função e algumas funções padrão, então isso é somente possível via a rotina de diferenciação do sistema. Assim, o posterior tem que aprender a controlar os símbolos novos.

Para esse propósito, MuPAD provê o tipo de domínio **DOM_FUNC_ENV** (abreviação para: function environment). Realmente, todas as funções matemáticas embutidas padrão são desse tipo na ordem para habilitar **float**, **diff**, **expand** etc.controlar então:

```
>> domtype(sin)
```

DOM_FUNC_ENV

Você pode chamar uma função ambiente como qualquer função "normal" ou procedimento:

```
>> sin(1.7)
```

0.9916648105

Uma função ambiente consiste de três operandos. O primeiro operando é um procedimento que computa o valor retornado de uma chamada de função. O segundo operando é um procedimento para imprimir uma chamada de função simbólica na tela. O terceiro operando é uma tabela contendo informação como as funções do sistema **float**, **diff**, **expand**, etc. devem controlar chamadas de funções simbólicas.

Você pode olhar no procedimento para avaliar a chamada de função simbólica com a função **expose**, há pouco igual para funções normais:

```
>> expose(sin)
proc(x)
  name sin;
  local f, y;
  option noDebug;
begin
  if args(0) = 0 then
    error("no arguments given")
  else
  ...
end_proc
```

Para manter o exemplo gerenciável, nós vamos escolher duas funções de perto relatadas funções não implementadas no muPAD (através delas podem ser representadas no MuPAD usando **hypergeom**). Nossos exemplos de funções são as funções integrais elípticas completas do primeiro e do segundo gênero, $K(z)$ e $E(z)$. Desde que MuPAD já tem um identificador predefinido **E**, nós vamos implementar essas funções como **ellipticE** e, por consistência, **ellipticK**. Essas funções aparecem em tais contextos diferentes como calculando o perímetro de uma elipse, o potencial gravitacional ou eletroestático de um anel uniforme ou a probabilidade que uma caminhada aleatória em três dsimensões sempre vai através da origem. Para essa representação, deixe-nos concentrar nas seguintes propriedades das funções E e K:

$$E'(z) = \frac{E(z) - K(z)}{2\,z}$$
$$K'(z) = \frac{E(z) - (1-z)K(z)}{2\,(1-z)z}$$

$$E(0) = K(0) = \frac{\pi}{2} \qquad E(1) = 1$$
$$K\left(\frac{1}{2}\right) = \frac{8\,\pi^{3/2}}{\Gamma\left(-\frac{1}{4}\right)^2} \qquad K(-1) = \frac{\Gamma\left(\frac{1}{4}\right)^2}{4\sqrt{2\pi}}$$

Isso é, nós estamos indo para implementar a derivada de E e K com as relações acima e faz as funções avaliar em pontos especiais. Adicionalmente, nós vamos implementar as funções avaliadas de tal um modo que eles são escritos como E e K na saída.

As funções básicas são fáceis de escrever:

```
>> ellipticE :=
    proc(x) begin
      if x = 0 then PI/2
      elif x = 1 then 1
      else procname(x) end_if
    end_proc:
>> ellipticK :=
    proc(x) begin
      if x = 0 then PI/2
      elif x = 1/2 then 8*PI^(3/2)/gamma(-1/4)^2
      elif x = -1 then gamma(1/4)^2/4/sqrt(2*PI)
      else procname(x) end_if
    end_proc:
```

Desde que os valores das funções são conhecidos apenas em lugares específicos, nós usamos **procname** (Seção 18.3) para retornar as expressões simbólicas **ellipticE (x)** e **EllipticK (x)**, respectivamente, para todos os argumentos onde os valores da função não são conhecidos. Isso rende:

```
>> ellipticE(0), ellipticE(1/2),
   ellipticK(12/17), ellipticK(x^2+1)
```

$$\frac{\pi}{2},\ \text{ellipticE}\left(\frac{1}{2}\right),\ \text{ellipticK}\left(\frac{12}{17}\right),\ \text{ellipticK}\left(x^2+1\right)$$

Você gera uma nova função ambiente pelos sentidos de **funcenv**:

```
>> output_E := f -> hold(E)(op(f)):
   ellipticE := funcenv(ellipticE, output_E):
>> output_K := f -> hold(K)(op(f)):
   ellipticK := funcenv(ellipticK, output_K):
```

Esses comandos convertem os procedimentos **ellipticE e ellipticK** para ambientes função. Os primeiros argumentos são os procedimentos para avaliação. O segundo argumento (opcional) de **funcenv** é o procedimento para a saída da tela. Nós queremos uma expressão simbólica **ellipticE (x)** mostrada como **E (x)** e igualmente **ellipticK (x)** deve ser mostrado como **K (x)**. Isso é alcançado pelo segundo argumento de **funcenv**, o qual você deve interpretar como uma rotina de conversão. Na entrada **ellipticE (x)**, isso retorna os objetos do MuPAD para imprimir na tela ao invés de **ellipticE (x)**. O argumento **f**, o qual representa **ellipticE (x)**, é convertido para a chamada de função não avaliada **E (x)** (note que **x=op (f)** para **f=ellipticE (x)** e que você precisa **hold (E)** para prevenir avaliação do identificador **E**). O sistema saiu essa expressão ao invés de **ellipticE(x)** na tela:

```
>> ellipticE(0), ellipticE(1/2),
   ellipticK(12/17), ellipticK(x^2+1)
```

$$\frac{\pi}{2},\ E\left(\frac{1}{2}\right),\ K\left(\frac{12}{17}\right),\ K\left(x^2+1\right)$$

O terceiro argumento (opcional) para **funcenv** é a tabela de *atributos de função*. Isso conta as funções do sistema **float, diff, expand** etc. como controlar chamadas simbólicas da forma **ellipticK (x)** e **ellipticE (x)**. No exemplo acima, nós não providenciamos tal qualquer atributos de função. Consequentemente, as funções do sistema ainda não conhecem como proceder e, por padrão, retornam a expressão ou a si mesmas simbolicamente

```
>> float(ellipticE(1/3)), expand(ellipticE(x + y)),
   diff(ellipticE(x), x), diff(ellipticK(x), x)
```

$$E\left(\frac{1}{3}\right),\ E\left(x+y\right),\ \frac{\partial}{\partial x}E\left(x\right),\ \frac{\partial}{\partial x}K\left(x\right)$$

Nomeando para a abertura "**diff**" de nossos ambientes de função, nós jogamos os atributos para a rotina de diferenciação **diff**:

```
>> ellipticE::diff :=
    proc(f,x)
     local z;
    begin
     z := op(f);
     (ellipticE(z) - ellipticK(z))/(2*z) * diff(z, x)
    end_proc:
>> ellipticK::diff :=
    proc(f,x)
     local z;
    begin
     z := op(f);
     (ellipticE(z) - (1-z)*ellipticK(z))/
        (2*(1-z)*z) * diff(z, x)
    end_proc:
```

Esses comandos contam **diff** que **diff (f, x)** com uma chamada de função simbólica **f = ellipticE (z)**, onde **z** depende em **x**, deve aplicar o procedimento denominado para **ellipticE::diff**. A bem conhecida regra de corrente rende

$$\frac{d}{dx}E(z) = E'(z)\frac{dz}{dx}$$

O procedimento específico implementa essa regra, onde a função entrada na expressão **f = ellipticE (z)** é dada por **z = op (f)**. *Agora MuPAD conhece as derivadas das funções representadas pelos identificadores **ellipticE** e **ellipticK***. Nós já implementamos as saídas da tela:

```
>> diff(ellipticE(z), z), diff(ellipticE(y(x)), x);
   diff(ellipticE(x*sin(x)), x)
```

$$\frac{E(z)-K(z)}{2z},\ \frac{(E(y(x))-K(y(x)))\ \frac{\partial}{\partial x}y(x)}{2y(x)}$$

$$\frac{(E(x\sin(x))-K(x\sin(x)))\ (\sin(x)+x\cos(x))}{2x\sin(x)}$$

Tão longe como **diff** está preocupado, a implementação de nossas duas integrais elípticas está completa agora:

```
>> diff(ellipticE(x), x, x)
```

$$\frac{\frac{E(x)-K(x)}{2x}+\frac{E(x)+K(x)(x-1)}{2x(x-1)}}{2x}-\frac{E(x)-K(x)}{2x^2}$$

```
>> normal(diff(ellipticK(2*x + 3), x, x, x))
   - (73 E(2 x + 3) + 86 K(2 x + 3) + 115 x E(2 x + 3) +

                                 2
     228 x K(2 x + 3) + 46 x  E(2 x + 3) +

          2                        3
     202 x  K(2 x + 3) + 60 x  K(2 x + 3)) /

          6         5         4          3          2
     (32 x  + 240 x  + 744 x  + 1220 x  + 1116 x  +

     540 x + 108)
```

Como uma aplicação, nós agora queremos MuPAD para computar os primeiros termos da expansão de Taylor da completa integral elíptica do primeiro gênero em volta de $x = 0$. Nós podemos usar a função **Taylor** desde que essas chamadas **diff** internamente:

```
>> taylor(ellipticK(x), x = 0, 6)
```

$$\frac{\pi}{2}+\frac{\pi x}{8}+\frac{9\pi x^2}{64}+\frac{99\pi x^3}{256}+\frac{11163\pi x^4}{4096}+\frac{855519\pi x^5}{16384}+O(x^6)$$

Exercício 18.1: Estenda a definição de **ellipticE** e **ellipticK** por uma abertura "**float**". Use **hypergeom::float** e as seguintes equivalências:

$$E(z) = \frac{\pi}{2}\,\mathtt{hypergeom}\left(\left[-\tfrac{1}{2},\tfrac{1}{2}\right],[1],z\right)$$
$$K(z) = \frac{\pi}{2}\,\mathtt{hypergeom}\left(\left[\tfrac{1}{2},\tfrac{1}{2}\right],[1],z\right)$$

Também, estenda as definições das funções tais que sua nova avaliação flutuante está automaticamente usada quando um valor de ponto flutuante é dado como entrada.

Exercício 18.2: Programe um valor absoluto em função **AbsN** como uma função ambiente. A chamada **Abs (x)** deve retornar o valor absoluto para números reais **x** do tipo de domínio **DOM_INT**, **DOM_RAT**, ou **DOM_FLOAT**. Para todos os outros argumentos, a saída simbólica **|x|** deve aparecer na tela. O valor absoluto é diferençável em $\mathbb{R}\{0\}$. Sua derivada é

$$\frac{d\,|y|}{d\,x} = \frac{|y|}{y}\frac{dy}{dx}$$

Jogue o atributo "**diff**" de acordo e compute a derivada de **Abs (x^3)**. Compare seu resultado para a derivada correspondente da função do sistema **abs**.

18.13 Um exemplo de programação: Diferenciação

Nesta seção, nós discutimos um exemplo demonstrando o funcionamento típico de um procedimento simbólico no MuPAD. Nós implementamos uma rotina de diferenciação simbólica que computa a derivada de expressões algébricas compostas de adições, multiplicações, exponenciações, algumas funções matemáticas (exp, ln, sin, cos, ...), constantes, e identificadores simbólicos.

Este exemplo é apenas para propósitos de ilustração desde que MuPAD já provê uma tal função: a rotina **diff**. Esta função é implementada no kernel do MuPAD e por essa razão muito rápido. Assim, uma função definida pelo usuário que é escrita na linguagem de programação do MuPAD pode dificilmente concluir a eficiência de **diff**.

As seguintes regras para diferenciação algébrica são válidas para a classe de expressões que nós consideramos:

(1) $\frac{d\,f}{dx} = 0$, se f não depende de x,

(2) $\frac{dx}{dx} = 1$,

(3) $\frac{d\,(f+g)}{dx} = \frac{d\,f}{dx} + \frac{dg}{dx}$ (linearidade),

(4) $\frac{d\,a\,b}{dx_b} = \frac{d}{dx}b + a\,\frac{db}{dx}$ (regra do produto),

(5) $\frac{d\,a}{dx} = \frac{d}{dx}\,e^{b\ln(a)} = e^{b\ln(a)}\,\frac{d}{dx}\,b\ln(a))$

$= a^b\ln(a)\,\frac{db}{dx} + a^{b-1}\,b\,\frac{da}{dx}$,

(6) $\frac{d}{dx} F(y(x)) = F'(y)\, \frac{dy}{dx}$ (regra de cadeia)

Além disso, para algumas funções *F*, a derivada é conhecida, e nós queremos levar isso em conta em nossa implementação. Para uma função desconhecida *F*, nós retornamos uma chamada de função simbólica da rotina da diferenciação.

O procedimento **Diff** implementa as propriedades acima na ordem declarada. Isso é chamada sintaxe **Diff (expression, identifier):**

```
>> Diff := proc(f, x : DOM_IDENT)                    // (0)
   local a, b, F, y; begin
      if not has(f, x) then return(0) end_if;        // (1)
      if f = x then return(1) end_if;                // (2)
      if type(f) = "_plus" then
         return(map(f, Diff, x)) end_if;             // (3)
      if type(f) = "_mult" then
         a := op(f, 1); b := subsop(f, 1 = 1);
         return(Diff(a, x)*b + a*Diff(b, x))         // (4)
      end_if;
      if type(f) = "_power" then
         a := op(f, 1); b := op(f, 2);
         return(f*ln(a)*Diff(b, x)
                + a^(b - 1)*b*Diff(a, x))            // (5)
      end_if;
      if op(f, 0) <> FAIL then
         F := op(f, 0); y := op(f, 1);               // (6)
         if F = hold(exp) then
            return( exp(y)*Diff(y, x)) end_if;       // (6)
         if F = hold(ln)  then
            return(  1/y  *Diff(y, x)) end_if;       // (6)
         if F = hold(sin) then
            return( cos(y)*Diff(y, x)) end_if;       // (6)
         if F = hold(cos) then
            return(-sin(y)*Diff(y, x)) end_if;       // (6)
         /* specify further known functions here */
      end_if;
      procname(args())                               // (7)
   end_proc:
```

Em (0), nós programamos um tipo de checagem automático do segundo argumento, o qual deve ser um identificador simbólico do tipo de domínio **DOM_IDENT.**

Linearidade da diferenciação é implementado em (3) pelos sentidos da função do MuPAD **map**:

```
>> map(f1(x) + f2(x) + f3(x), Diff, x)
```

$$\text{Diff}(\text{f1}(x), x) + \text{Diff}(\text{f2}(x), x) + \text{Diff}(\text{f3}(x), x)$$

Em (4), nós controlamos a expressão do produto $f = f_1 . f_2 . \ldots$: o comando **a:=op (f, 1)** determina o primeiro fator $a = f_1$, então **subsop (f, 1=1)** substitui este fator por 1, tal que *b* assume o valor $f_2 . f_3 . \ldots$ por si mesmo um produto, isso leva a outra execução do passo (4) no próximo nível de recursão. Deste modo, (4) controla produtos de comprimento arbitrário.

Passo (5) diferencia potências. Para $f = a^b$, a chamada **op (f, 1)** retorna a base *a* e **op (f, 2)** o expoente *b*. Em particular, isso cobre todas a expressões monomiais da forma $f = x^n$ para constante *n*. As chamadas recursivas para **Diff (b, x)=0**, respectivamente, e a expressão retornada em (5) simplifica para o resultado correto $x^{n-1}\, n$.

Se a expressão *f* é uma chamada de função simbólica da forma $f = F(y)$, nós extraímos a função "exterior" *F* em (6) via **F:=op (f, 0)** (do contrário, **F** consegue o valor **FAIL**). Próximo, nós controlamos o caso onde *F* é uma função com um argumento *y* e extraímos o produto "interno" por **y:=op (f, 1).** Se *F* é o nome de uma função com derivada conhecida (tal como *F* = exp, ln, sin, cos), então nós aplicamos a regra em cadeia. Isso é fácil para estender essa lista de funções *F* com derivadas conhecidas. Em particular, você pode adicionar uma fórmula para diferenciar expressões simbólicas da forma **int(**. , .). Extensões para funções *F* com mais que um argumento também são possíveis.

Finalmente, passo (7) retorna **Diff (f, x)** simbolicamente se não tem simplificações da expressão *f* acontecida nos passos (1) por (6).

O modo de **Diff** da operação é adotado do sistema de função **diff**. Compare o seguinte resultado para esses retornados por **diff**:

```
>> Diff(x*ln(x + 1/x), x)
```

$$\ln\left(x + \frac{1}{x}\right) - \frac{x\left(\frac{1}{x^2} - 1\right)}{x + \frac{1}{x}}$$

```
>> Diff(f(x)*sin(x^2), x)
```

$$\sin\left(x^2\right)\, \text{Diff}\left(f(x), x\right) + 2\, x\, f(x)\, \cos\left(x^2\right)$$

18.14 Programando exercícios

Exercício 18.3: Escreve um procedimento curto **date** que toma três integrais **mês, dia, ano** como entrada e imprime a data no modo usual. Por exemplo, a chamada **data** (5, 3, 1990) deve render a saída de tela 5/3/1990.

Exercício 18.4: Nós definimos a função f: $\mathbb{N} \to \mathbb{N}$ por

$$f(x) = \begin{cases} 3x + 1 \text{ para } x \text{ ímpar}, \\ x/2 \text{ para } x \text{ par} \end{cases}$$

O "problema $(3x + 1)$" pergunta se para um valor arbitrário inicial $x_0 \in \mathbb{N}$, a sequência recursivamente definida por $x_{i+1} := f(x_i)$ contém o valor 1. Escrever um programa que na entrada x_0 retorna o menor índice i com $x_i = 1$.

Exercício 18.5: Implemente uma função **Gcd** para computar o máximo divisor comum de duas integrais positivas.Certamente, você não deve usar as funções do sistema **gcd** e **igcd**. Dica: o algoritmo euclidiano para computar o gcd é baseado na observação

gcd (a, □) = gcd(a mod b, b) = gcd (b, a mod b)

e os fatos gcd(0, b) = b.

Exercício 18.6: Implemente a função **Quadrature**. Para uma função **f** e uma lista no MuPAD **X** para valores numéricos

$x_0 < x_1 < x_n < \cdots < x_n$,

A chamada **Quadrature (f, X)** deve computar uma aproximação numérica da integral

$$\int_{x_0}^{x_n} f(x)\,dx \approx \sum_{i=0}^{n-1} (x_{i+1} - x_i)\, f(x_i).$$

Exercício 18.7: O método de Newton para encontrar uma raiz numérica de uma função f: $\mathbb{R} \to \mathbb{R}$ emprega a iteração $x_{i+1} = F(x_i)$, onde $F(x) = x - f(x)/(f^{\wedge\prime}(x))$. Escreva um procedimento **Newton**. A chamada **Newton (f, x0, n)**, com uma expressão **f**, deve retornar os primeiros $n + 1$ elementos $x_0 \ldots, x_n$ da uma sequ~encia de Newton.

Exercício 18.8: O *triangulo de Sierpinski* é um fractal bem conhecido. Nós definimos uma variante dele como se segue. O triângulo de Sierpinski é um conjunto de todos os pontos $(x, y) \in \mathbb{N} \times \mathbb{N}$ com a seguinte propriedade: ali existe ao menos uma posição nas expansões binárias de x e y tais que ambas tem 1-bit nessa posição. Escreva um programa **Sierpinski** que na entrada **xmax, ymax** plota o conjunto de todos tais pontos com integrais coordenadas no alcance $1 \le x \le$ **xmaxN,** $1 \le y \le$ **ymax.** Dica: a função **numlib::g_adic** computa a expansão binária de uma integral. O gráfico primitivo **plot::PointList2d** pode ser usado para criar um plot dos pontos.

Exercício 18.9: Uma *fórmula lógica* é composta de identificadores e os operadores **and**, **or,** e **not**. Por exemplo:

```
>> formula := (x and y) or
              ( (y or z) and (not x) and y and z )
```

Tal uma fórmula é chamada *satisfatória* se essa é possível de nomear os valores **TRUE** e **FALSE** para todos os identificadores em um caminho parecido que a fórmula pode ser avaliada para **TRUE**. Escrever um programa que checa se uma fórmula lógica arbitrária é satisfatória.

Capítulo 19

Soluções dos Exercícios

Exercício 2.1: A página de ajuda ?diff diz para você como computar derivadas de alta ordem:

```
>> diff(sin(x^2), x, x, x, x, x)
   32 x^5 cos(x^2) - 120 x cos(x^2) + 160 x^3 sin(x^2)
```

Você também pode usar o comando mais extenso diff(diff(...,x),x).

Exercício 2.2: As representações exatas são:

```
>> sqrt(27) - 2*sqrt(3), cos(PI/8)
   sqrt(3), sqrt(sqrt(2) + 2)/2
```

As aproximações numéricas são:

```
>> DIGITS := 5:
>> float(sqrt(27) - 2*sqrt(3)), float(cos(PI/8))
   1.7321, 0.92388
```

Elas estão corretos por dentro de 5 dígitos.

Exercício 2.3:

```
>> expand((x^2 + y)^5)
```
$$x^{10} + 5\,x^8\,y + 10\,x^6\,y^2 + 10\,x^4\,y^3 + 5\,x^2\,y^4 + y^5$$

Exercício 2.4:

```
>> normal((x^2 - 1)/(x + 1))
```
$$x - 1$$

Exercício 2.5: Você pode plotar a função singular $f(x) = 1/\sin(x)$ no intervalo [1, 10] sem qualquer problema desde que o (eqüidistante) gráfico exemplifique pontos neste intervalo não contendo singularidades:

```
>> plotfunc2d(1/sin(x), x = 1..10)
```

Contudo, no intervalo [0, 10], você marcha dentro de problemas. Limites pertencem aos pontos de amostra. Um erro ocorre quando o sistema tenta igualar 1/sin(0)=1/0 no limite à esquerda.

Exercício 2.6: MuPAD imediatamente retorna os limites clamados:

```
>> limit(sin(x)/x, x = 0),
   limit((1 - cos(x))/x, x = 0),
   limit(ln(x), x = 0, Right)
```
$$1,\ 0,\ -\infty$$
```
>> limit(x^sin(x), x = 0),
   limit((1 + 1/x)^x, x = infinity),
   limit(ln(x)/exp(x), x = infinity)
```
$$1,\ e,\ 0$$
```
>> limit(x^ln(x), x = 0, Right),
   limit((1 + PI/x)^x, x = infinity),
   limit(2/(1 + exp(-1/x)), x = 0, Left)
```
$$\infty,\ e^{\pi},\ 0$$

O resultado **undefined** denota um limite não-existindo:

```
>> limit(exp(cot(x)), x = 0)
```
undefined

Exercício 2.7: Você pode obter o primeiro resultado na forma desejada fatorando:

```
>> sum(k^2 + k + 1 , k = 1..n): % = factor(%)
```
$$\frac{n^3}{3} + n^2 + \frac{5\,n}{3} = \left(\frac{1}{3}\right) \cdot n \cdot \left(n^2 + 3\,n + 5\right)$$
```
>> sum((2*k - 3)/((k + 1)*(k + 2)*(k + 3)),
      k = 0..infinity)
```
$$-\frac{1}{4}$$
```
>> sum(k/(k - 1)^2/(k + 1)^2, k = 2..infinity)
```
$$\frac{5}{16}$$

Exercício 2.8:

```
>> A := matrix([[1, 2, 3], [4, 5, 6], [7, 8, 0]]):
>> B := matrix([[1, 1, 0], [0, 0, 1], [0, 1, 0]]):
>> 2*(A + B), A*B
```
$$\begin{pmatrix} 4 & 6 & 6 \\ 8 & 10 & 14 \\ 14 & 18 & 0 \end{pmatrix},\ \begin{pmatrix} 1 & 4 & 2 \\ 4 & 10 & 5 \\ 7 & 7 & 8 \end{pmatrix}$$
```
>> (A - B)^(-1)
```
$$\begin{pmatrix} -\frac{5}{2} & \frac{3}{2} & -\frac{5}{7} \\ \frac{5}{2} & -\frac{3}{2} & \frac{6}{7} \\ -\frac{1}{2} & \frac{1}{2} & -\frac{2}{7} \end{pmatrix}$$

Exercício 2.9: a) A função **numlib::mersenne** retorna uma lista de valores para p rendendo os 41 atualmente conhecidos primos de Mersenne que tem sido encontrados em supercomputadores. A computação atual para $1 < p \leq 1000$ pode ser facilmente interpretada no MuPAD:

```
>> select([$ 1..1000], isprime):
>> select(%, p -> (isprime(2^p - 1)))
```
Depois de algum tempo você obtém a lista de valores desejados de p:

$$[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607]$$

Os primos correspondentes de Mersenne são:

```
>> map(%, p -> (2^p-1))
   [3, 7, 31, 127, 8191, 131071, 524287, 2147483647,

      2305843009213693951, 618970019642690137449562111,

      162259276829213363391578010288127, ... ]
```

b) Dependendo da velocidade do seu computador, você só pode testar os primeiros 11 ou 12 números de Fermat em uma quantidade razoável de tempo. Note que o 12º número de Fermat já tem 1234 dígitos.

```
>> Fermat := n -> (2^(2^n) + 1): isprime(Fermat(10))
    FALSE
```

Apenas os primos conhecidos de Fermat estão nos primeiros cinco números de Fermat (incluindo **Fermat (0))**. Verdadeiramente, se MuPAD testa os primeiros 12 números de Fermat, então depois de algum tempo isso retorna os seguintes cinco valores:

```
>> select([Fermat(i) $ i = 0..11], isprime)
    [3, 5, 17, 257, 65537]
```

Exercício 4.1: O primeiro operando de um a potência é a base, o segundo é o expoente. O primeiro e o segundo operando de uma equação são o lado esquerdo e o lado direito, respectivamente. Os operandos de uma função chamada são seus argumentos:

```
>> op(a^b, 1), op(a^b, 2)
    a, b
>> op(a = b, 1), op(a = b, 2)
    a, b
>> op(f(a, b), 1), op(f(a, b), 2)
    a, b
```

Exercício 4.2: A lista com as duas equações é **op (set, 1).** Esse segundo operando é a equação **y=...,** da qual segundo operando é o sentido do lado direito:

```
>> set := solve({x+sin(3)*y = exp(a),
                 y-sin(3)*y = exp(-a)}, {x,y})
    {[x = (sin(3) e^-a - e^a + sin(3) e^a)/(sin(3) - 1), y = -e^-a/(sin(3) - 1)]}
>> y := op(set, [1, 2, 2])
    -e^-a/(sin(3) - 1)
```

Use **assign(op(set))** para executar tarefas de ambas as incógnitas **x** e **y** simultaneamente.

Exercício 4.3: Se pelo menos um número em uma expressão numérica é um número de pontos flutuantes, então o resultado é um número em ponto flutuante:

```
>> 1/3 + 1/3 + 1/3, 1.0/3 + 1/3 + 1/3
    1, 1.0
```

Exercício 4.4: Você obtém o desejado número de pontos flutuantes imediatamente aplicando o comando **float:**

```
>> float(PI^(PI^PI)), float(exp(PI*sqrt(163)/3))
    1.340164183 · 10^18, 640320.0
```

Note que apenas os primeiros 10 dígitos destes valores são confiáveis desde que essa é a precisão padrão. Realmente, para valores extensos de **DIGITS**, você encontra:

```
>> DIGITS := 100:
>> float(PI^(PI^PI)), float(exp(PI*sqrt(163)/3))
   1340164183006357435.29744912964013141509937497457349\
   9
   2377879275165860340926190940681482694726113011 42

      ,

   640320.00000000060486373504901603947174181881853947\
   5
   7148576036659181946522182582869425363408158226 46
```

Nós computamos 235 dígitos decimais de **PI** para obter o 234º dígito correto depois do ponto decimal. Depois ajustamos **DIGITS:=235**, o resultado é o último dígito mostrado de **float (PI).** Um modo mais elegante é multiplicar por 10^{234}. Então o dígito desejado é o primeiro dígito antes do ponto decimal, e nós o obtemos truncando os dígitos depois do ponto decimal:

```
>> DIGITS := 235: trunc(10^234*PI) - 10*trunc(10^233*PI)
    6
```

Exercício 4.5: a) Internamente, MuPAD computa exatamente com alguns dígitos adicionais não mostrados na saída.

```
>> DIGITS := 10: x := 10^50/3.0; floor(x)
```

$$3.333333333 \cdot 10^{49}$$

$$33333333333333333332837742572648668761603891527680$$

Depois de acrescentar o comando **DIGITS**, MuPAD mostra os dígitos adicionais. Contudo, nem todos eles estão corretos: Reiniciando a computação com o valor acrescentado de **DIGITS** para obter a precisão desejada:

```
>> DIGITS := 40: x := 10^50/3.0
```

$$3.3333333333333333333333333333333333333 \cdot 10^{49}$$

Exercício 4.6: Os nomes **caution!-!**, **x-y**, e **Jack&Jill** são inválidos desde que eles contém os caracteres especiais **!**, **-**, e **&**, respectivamente. Desde que um nome de um identificador não deva começar com um número, 2x não é tampouco válido. Os nomes **diff** e **exp** são nomes válidos de identificadores. Contudo, você não pode determinar valores para eles desde que eles são nomes protegidos de funções do MuPAD.

Exercício 4.7: Nós usamos o operador $ (Seção 4.5) para gerar o conjunto de equações e o conjunto de incógnitas. Então a chamada **solve** retorna um conjunto de equações mais simples:

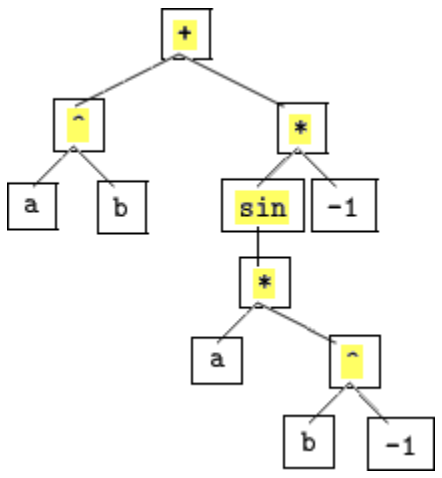
```
>> equations := {(x.i + x.(i+1) = 1) $ i = 1..19,
                 x20 = PI}:
>> unknowns := {x.i $ i = 1..20}:
>> solutions := solve(equations, unknowns)
   {[x1 = 1 - PI, x10 = PI, x11 = 1 - PI, x12 = PI,

     x13 = 1 - PI, x14 = PI, x15 = 1 - PI, x16 = PI,

     x17 = 1 - PI, x18 = PI, x19 = 1 - PI, x2 = PI,

     x20 = PI, x3 = 1 - PI, x4 = PI, x5 = 1 - PI,

     x6 = PI, x7 = 1 - PI, x8 = PI, x9 = 1 - PI]}
```

Nós usamos a função **assign** para determinar os valores computados para os identificadores:

```
>> assign(op(solutions, 1)): x1, x2, x3, x4, x5, x6
```

$$1 - \pi,\ \pi,\ 1 - \pi,\ \pi,\ 1 - \pi,\ \pi$$

Exercício 4.8: MuPAD armazena a expressão **a^b-sin(a/b)** na forma **a^b+(-1)*sin(a*b^(-1))**. Esta é a árvore expressão:



Exercício 4.9: Nós observamos que:

```
>> op(2/3); op(x/3)
```

$$2,\ 3$$

$$x,\ \frac{1}{3}$$

A razão é que 2/3 é do tipo de domínio **DOM_RAT,** o qual os operandos são o numerador e o denominador. O tipo de domínio da expressão simbólica **x/3** é **DOM_EXPR** e sua representação interna é **x*(1/3)**. A situação é similar para 1 + 2 * I e x + 2 * I:

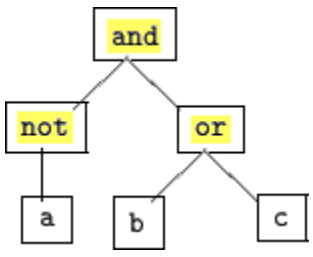
```
>> op(1 + 2*I); op(x + 2*I)
```

$$1,\ 2$$

$$x,\ 2\mathrm{i}$$

O primeiro objeto é do tipo de domínio **DOM_COMPLEX**. Seus operandos são a parte real e imaginária. Os operandos da expressão simbólica **x + 2 * I** são o primeiro e o segundo termo da soma.

Exercício 4.10: A árvore da expressão da condição = (not a) e (b ou c) é:



Dessa forma **op (condition, 1)=not a** e **op (condition, 2)=b ou c**. Nós obtemos os núcleos a, b, c como segue:

```
>> op(condition, [1, 1]), op(condition, [2, 1]),
   op(condition, [2, 2])
```

$$a,\ b,\ c$$

Exercício 4.11: Você pode usar ambas a função tarefa **_assign** apresentada na Seção **4.3** e o operador tarefa **:=**.

```
>> _assign(x.i, i) $ i = 1..100:
>> (x.i := i) $ i = 1..100:
```

Você também pode passar um conjunto de transferência de equações para a função **assign**:

```
>> assign({x.i = i $ i = 1..100}):
```

Exercício 4.12: Desde que uma seqüência é um argumento válido do operador seqüência, você pode executar o resultado desejado como segue:

```
>> (x.i $ i) $ i = 1..10
   x1, x2, x2, x3, x3, x3, x4, x4, x4, x4, ...
```

Exercício 4.13: Nós usamos a função adicional **_plus** e gera seu argumento seqüencial via **$**:

```
>> _plus(((i+j)^(-1) $ j = 1.. i) $ i=1..10)
```

$$\frac{1464232069}{232792560}$$

Exercício 4.14:

```
>> L1 := [a, b, c, d]: L2 := [1, 2, 3, 4]:
>> L1.L2, zip(L1, L2, _mult)
```

$$[a, b, c, d, 1, 2, 3, 4],\ [a, 2\,b, 3\,c, 4\,d]$$

Exercício 4.15: A função **_mult** multiplica seus argumentos:

```
>> map([1, x, 2], _mult, multiplier)
```

$$[\text{multiplier}, x\ \text{multiplier}, 2\,\text{multiplier}]$$

Nós usamos **map** para aplicar a função **sublist ->map (sublist, _mult, 2)** para as sublistas em uma lista abrigada:

```
>> L := [[1, x, 2], [PI], [2/3, 1]]:
   map(L, map, _mult, 2)
```

$$\left[[2, 2\,x, 4], [2\,\pi], \left[\frac{4}{3}, 2\right]\right]$$

Exercício 4.16: Para:

```
>> X := [x1, x2, x3]: Y := [y1, y2, y3]:
```

Os produtos são dados imediatamente por

```
>> _plus(op(zip(X, Y, _mult)))
```

$$x1\,y1 + x2\,y2 + x3\,y3$$

A seguinte função **f** multiplica cada elemento da lista **Y** por seu parâmetro de entrada **x** e retorna a lista resultante:

```
>> f := x -> (map(Y, _mult, x)):
```

O próximo comando substitui cada elemento de **X** pela lista retornada por **f:**

```
>> map(X, f)
```

$$[[x1\,y1, x1\,y2, x1\,y3], [x2\,y1, x2\,y2, x2\,y3], [x3\,y1, x3\,y2, x3\,y3]]$$

Exercício 4.17: Para cada *m*, nós usamos o gerador de seqüência **$** para criar uma lista de todas as integrais para serem verificadas. Então nós extraímos todos os números primos da lista via **select (., isprime)**. O número de primos é somente **nops** da lista resultante. Nós computamos este valor para todo *m* entre 0 e 41:

```
>> nops(select([(n^2 + n + m) $ n = 1..100], isprime))
     $ m = 0..41
   1, 32, 0, 14, 0, 29, 0, 31, 0, 13, 0, 48, 0, 18, 0,

      11, 0, 59, 0, 25, 0, 14, 0, 28, 0, 28, 0, 16, 0,

      34, 0, 35, 0, 11, 0, 24, 0, 36, 0, 17, 0, 86
```

Há uma explicação simples para os valores zero para igualar $m > 0$. Desde que $n^2 + n = n(n - 1)$ é sempre igual, $n2 + n + m$ é uma mesma integral maior que 2 e conseqüentemente não é um primo.

Exercício 4.18: Nós armazenamos a criança em uma lista C e removemos aquele que foi contado fora no fim de cada rodada. Nós representamos as posições 1,2,..., *n* de *n* crianças por uma lista com as integrais correspondentes. Deixe **fora** ∈ {1,..., *n}* ser a posição da última criança que ficou fora da contagem. Depois de apagar o enésimo elemento, a próxima volta começa na posição out na lista abreviada. No fim da volta, depois das *m* palavras, a criança na posição **out + m – 1** na lista corrente está fora da contagem. Desde que nós estamos contando de modo cíclico, nós tomamos este módulo de valor o número de crianças restantes. Note, contudo, que **a mod b** produz números no alcance 0, 1,..., b – 1 particularmente que 1,..., b. Isto é sujeito pelo uso **((a – 1) mod b) + 1** ao invés de **a mod b:**

```
>> m := 8: n := 12: C := [$ 1..n]: out := 1:
>> out := ((out + m - 2) mod nops(C)) + 1:
>> C[out]
    8
>> delete C[out]:
>> out := ((out + m - 2) mod nops(C)) + 1:
>> C[out]
    4
```

etc. Isto é útil para implementar a conta por um loop (Capítulo 16):

```
>> m := 8: n := 12: C := [$ 1..n]: out := 1:
>> repeat
     out := ((out + m - 2) mod nops(C)) + 1:
     print(C[out]);
     delete C[out];
   until nops(C) = 0 end_repeat:
                              8

                              4

                              1

                              11

                             ...

                              9

                              5
```

Exercício 4.19: As duas conversões seguintes *list* ↣ *set* ↣ *list* em geral mudam a ordem dos elementos da lista:

```
>> set := {op(list)}:  list := [op(set)]:
```

Exercício 4.20:

```
>> A := {a, b, c}: B := {b, c, d}: C := {b, c, e}:
>> A union B union C, A intersect B intersect C,
   A minus (B union C)
    {a, b, c, d, e}, {b, c}, {a}
```

Exercício 4.21: Você obtém a união via **_union**:

```
>> M := {{2, 3}, {3, 4}, {3, 7}, {5, 3}, {1, 2, 3, 4}}:
>> _union(op(M))
    {1, 2, 3, 4, 5, 7}
```

E a intersecção via **_intersect**:

```
>> _intersect(op(M))
    {3}
```

Exercício 4.22: A função **combinate::subsets** (**M, k)** retorna uma lista de todos os subconjuntos com *k* elementos de um conjunto *M*:

```
>> M := {i $ i = 5..20}:
>> subsets := combinat::subsets(M, 3):
```

O número de tais subconjuntos é:

```
>> nops(subsets)
    560
```

A função **combinate::subsets::count** retorna o número de subconjuntos sem (expansivamente) gerando eles primeiro:

```
>> combinat::subsets::count(M, 3)
    560
```

Exercício 4.23:

```
>> telephoneDirectory := table(Ford = 1815,
     Reagan = 4711, Bush = 1234, Clinton = 5678):
```

Uma chamada indexada retorna o número de Ford:

```
>> telephoneDirectory[Ford]
    1815
```

Você pode usar **select** para extrair todas as entradas de tabela contendo o número 5678:

```
>> select(telephoneDirectory, has, 5678)
    [ Clinton = 5678
```

Exercício 4.24: O comando **[op(Table)]** retorna uma lista de todas as equações designadas. A chamada **map(. , op, i) (*i* = 1, 2)** extrai os sentidos dos lados esquerdo e direito, respectivamente, das equações:

```
>> T := table(a = 1, b = 2,
              1 - sin(x) = "derivative of x + cos(x)" ):
>> indices := map([op(T)], op, 1)
    [a, b, 1 − sin (x)]
>> values := map([op(T)], op, 2)
    [1, 2, "derivative of x + cos(x)"]
```

Exercício 4.25: Os seguintes momentos (em milissegundos) mostram que gerando uma tabela é mais tempo consumindo:

```
>> n := 100000:
>> time((T := table( (i=i) $ i=1..n))),
   time((L := [i $ i=1..n]))
    3750, 820
```

Contudo, trabalhando com tabelas é notavelmente fácil. As seguintes tarefas criam uma entrada de tabela adicional e estendem a lista por um elemento, respectivamente:

```
>> time((T[n + 1] := New)), time((L := L.[New]))
    10, 140
```

Exercício 4.26: Nós usamos o gerador de seqüência **$** para criar uma lista abrigada e passamos isso para **array:**

```
>> n := 20:
   array(1..n, 1..n,
     [[1/(i + j - 1) $ j = 1..n] $ i = 1..n]):
```

Exercício 4.27:

```
>> TRUE and (FALSE or not (FALSE or not FALSE))
    FALSE
```

Exercício 4.28: Nós usamos a função **zip** para gerar uma lista de comparações. Nós passamos a função de sistema **_less** como terceiro argumento, o qual gera inequações da forma **a** < **b**. Então nós extraímos a seqüência de inequações via **op** e passamos ela para **_and**:

```
>> L1 := [10*i^2 - i^3 $ i = 1..10]:
>> L2 := [i^3 + 13*i $ i = 1..10]:
>> _and(op(zip(L1, L2, _less)))
   9 < 14 and 32 < 34 and 63 < 66 and 96 < 116 and

      125 < 190 and 144 < 294 and 147 < 434 and

      128 < 616 and 81 < 846 and 0 < 1130
```

Finalmente, igualando esta expressão com **bool** responde a questão:

```
>> bool(%)
   TRUE
```

Exercício 4.29: A função **sort** não encurta os identificadores alfabeticamente pelos seus nomes mas de acordo a sua ordem interna (Seção 4.6). Assim, nós convertemos eles para séries via **expr2text** antes encurtando:

```
>> map(anames(All), expr2text)
   ["Ax", "Axiom", "AxiomConstructor", "C_", "Cat",

      "Category", ... , "zeta", "zip"]
```

Exercício 4.30: Nós computamos a reflexão do palíndromo

```
>> text := "Never odd or even":
```

Passando a seqüência refletida de caracteres individuais para a função **_concat**, a qual converte isso para uma série novamente:

```
>> n := length(text):
   _concat(text[n - i + 1] $ i = 1..n)
    "neve ro ddo reveN"
```

Isso pode ser realizado com a chamada **revert (text).**

Exercício 4.31:

```
>> f := x -> (x^2): g := x -> (sqrt(x)):
>> (f@f@g)(2), (f@@100)(x)
```

$$4,\ x^{1267650600228229401496703205376}$$

Exercício 4.32: A seguinte função faz o trabalho:

```
>> f := L -> [L[nops(L) + 1 - i] $ i = 1..nops(L)]
        L -> [L[(nops(L) + 1) - i] $ i = 1..nops(L)]

>> f([a, b, c])
```

$$[c, b, a]$$

Contudo, a solução mais simples é usar **f:=revert**.

Exercício 4.33: Você pode usar a função **last** (Capítulo 12) para gerar os polinômios de Chebyshev como expressões:

```
>> T0 := 1: T1 := x:
>> T2 := 2*x*% - %2; T3:= 2*x*% - %2; T4:= 2*x*% - %2
```

$$2x^2 - 1$$

$$2x\left(2x^2 - 1\right) - x$$

$$1 - 2x^2 - 2x\left(x - 2x\left(2x^2 - 1\right)\right)$$

Um modo muito elegante é traduzir a definição recursiva dentro de uma função do MuPAD que trabalha recursivamente:

```
>> T := (k, x) ->
       (if k < 2
           then x^k
           else 2*x*T(k - 1, x) - T(k - 2, x)
        end_if):
```

Então nós obtemos:

```
>> T(i, 1/3) $ i = 2..5
```

$$-\frac{7}{9},\ -\frac{23}{27},\ \frac{17}{81},\ \frac{241}{243}$$

```
>> T(i, 0.33) $ i = 2..5
```

$$-0.7822,\ -0.846252,\ 0.22367368,\ 0.9938766288$$

```
>> T(i, x) $ i = 2..5
     2              2
  2 x  - 1, 2 x (2 x  - 1) - x,

          2                    2
    1 - 2 x  - 2 x (x - 2 x (2 x  - 1)), x -

           2                               2
    2 x (2 x  - 1) - 2 x (2 x (x - 2 x (2 x  - 1)) +

       2
    2 x  - 1)
```

Você pode obter as representações expandidas dos polinômios inserindo uma chamada para **expand** (Seção 9.1) na definição da função. Os polinômios de Chebyshev já são implementados em **orthpoly**, a biblioteca para polinômios ortogonais. O i^o polinômio de Chebyshev é retornado pela chamada **orthpoly::chebyshev1 (i, x).**

Exercício 4.34: Em princípio, você pode computar a derivada de f no MuPAD e substituir $x = 0$. Mas isso é mais simples para aproximar a função pelas séries de Taylor as quais levam termos que descrevem o comportamento nas proximidades de $x = 0$:

```
>> taylor(tan(sin(x)) - sin(tan(x)), x = 0, 8)
```

$$\frac{x^7}{30} + \frac{29\,x^9}{756} + \frac{1913\,x^{11}}{75600} + \frac{95\,x^{13}}{7392} + O\left(x^{15}\right)$$

Assim, $f(x) = \frac{x^7}{30}(1 + O(x^2))$, e conseqüentemente f tem uma raiz de ordem 7 em $x = 0$.

Exercício 4.35: A razão para a diferença entre os resultados

```
>> taylor(diff(1/(1 - x), x), x);
   diff(taylor(1/(1 - x), x), x)
```

$$1 + 2\,x + 3\,x^2 + 4\,x^3 + 5\,x^4 + 6\,x^5 + O\left(x^6\right)$$

$$1 + 2\,x + 3\,x^2 + 4\,x^3 + 5\,x^4 + O\left(x^5\right)$$

É o picado determinado pela variável ambiente **ORDER** com o valor padrão 6. Ambas as chamadas **Taylor** computam as séries correspondentes acima para $O(x^6)$:

```
>> taylor(1/(1 - x), x)
```

$$1 + x + x^2 + x^3 + x^4 + x^5 + O\left(x^6\right)$$

A ordem do termo **0(x^5)** aparece quando o termo **0(x^6)** é diferenciada:

```
>> diff(%, x)
```

$$1 + 2\,x + 3\,x^2 + 4\,x^3 + 5\,x^4 + O\left(x^5\right)$$

Exercício 4.36: Uma expansão assintótica rende:

```
>> f := sqrt(x + 1) - sqrt(x - 1):
>> g := series(f, x = infinity)
```

$$\frac{1}{\sqrt{x}} + \frac{1}{8\sqrt[2]{x^5}} + \frac{7}{128\sqrt[2]{x^9}} + O\left(\frac{1}{\sqrt[2]{x^{11}}}\right)$$

Assim

$$f \approx \frac{1}{\sqrt{x}}\left(1 + \frac{1}{8x^2} + \frac{7}{128x^4} + \cdots\right),$$

E conseqüentemente $f(x) \approx 1/\sqrt{x}$ para todo $x \gg 1$. A melhor próxima aproximação é $f(x) \approx \frac{1}{\sqrt{x}}\left(1 + \frac{1}{8x^2}\right)$.

Exercício 4.37: O comando **?revert** pede a página de ajuda correspondente.

```
>> f := taylor(sin(x + x^3), x); g := revert(%)
```

$$x + \frac{5\,x^3}{6} - \frac{59\,x^5}{120} + O\left(x^7\right)$$

$$x - \frac{5\,x^3}{6} + \frac{103\,x^5}{40} + O\left(x^7\right)$$

Para checar este resultado, nós consideramos a composição de f e g, as quais a série expansão é que da função identidade $x \mapsto x$:

```
>> g@f
```

$$x + O\left(x^7\right)$$

Exercício 4.38: Nós executamos a computação que completa o anel coeficiente padrão. (Seção **4.15.1**), a qual cumpre ambos os números racionais e os números de pontos flutuantes:

```
>> n := 16:
>> H := matrix(n, n, (i, j) -> ((i + j -1)^(-1))):
>> e := matrix(n, 1, 1): b := H*e:
```

Nós computamos primeiro a solução do sistema de equações $H\vec{x} = \vec{b}$ com aritmética exata sobre os números racionais. Então nós convertemos todas as entradas de **H** e **b** para números de pontos flutuantes e resolvemos o sistema numericamente (e no modo instável e lento, usando um explicito ao invés de chamar **numeric::matlinsolve**):

```
>> exact = H^(-1)*b, numerical = float(H)^(-1)*float(b)
```

$$\text{exact} = \begin{pmatrix} 1 \\ 1 \\ \cdots \\ 1 \\ 1 \\ 1 \end{pmatrix},\ \text{numerical} = \begin{pmatrix} 0.999999993 \\ 1.000000164 \\ \cdots \\ -11.875 \\ 4.036132812 \\ 0.3975830078 \end{pmatrix}$$

Os erros na solução numérica originam de arrendondamento de erros. Para demonstrar isto, nós repetimos a computação numérica com precisão mais alta:

```
>> DIGITS := 20:
>> numerical = float(H)^(-1)*float(b)
```

$$\text{numerical} = \begin{pmatrix} 1.0000000505004147319 \\ 0.99999992752642441474 \\ \cdots \\ 0.99999889731407165527 \\ 1.0000003278255462646 \\ 0.9999999580904841423 \end{pmatrix}$$

Exercício 4.39: Nós olhamos os valores onde o determinante da matriz desaparece:

```
>> matrix([[1, a, b], [1, 1, c ], [1, 1, 1]]):
>> factor(linalg::det(%))
```

$$(c-1)\cdot(a-1)$$

A matriz é irreversível a menos em $a = 1$ ou $c = 1$.

Exercício 4.40: Nós primeiro armazenamos os dados da matriz em ordens. Estas ordens são usadas mais tarde para gerar matrizes acima de anéis coeficientes diferentes:

```
>> a := array(1..3, 1..3, [[ 1, 3, 0],
                          [-1, 2, 7],
                          [ 0, 8, 1]]):
>> b := array(1..3, 1..2, [[7, -1], [2, 3], [0, 1]]):
```

Para simplificar a seguinte chamada de função nós exportamos a biblioteca **Dom**:

```
>> export(Dom):
```

Agora nós definimos o construtor **MQ** para matrizes sobre os números racionais e convertemos as ordens para as matrizes correspondentes:

```
>> MQ := Matrix(Rational): A := MQ(a): B := MQ(b):
```

O método "**transpose**" do construtor computa a transposição de uma matriz **B** via **MQ::transpose(B)**:

```
>> (2*A + B*MQ::transpose(B))^(-1)
```

$$\begin{pmatrix} \frac{34}{1885} & \frac{7}{1508} & -\frac{153}{7540} \\ \frac{11}{3770} & -\frac{31}{3016} & \frac{893}{15080} \\ -\frac{47}{3770} & \frac{201}{3016} & -\frac{731}{15080} \end{pmatrix}$$

Computando sobre a classe resíduo anel módulo 7 nós encontramos:

```
>> Mmod7 := Matrix(IntegerMod(7)):
>> A := Mmod7(a): B := Mmod7(b):
>> C := (2*A + B*Mmod7::transpose(B)): C^(-1)
```

$$\begin{pmatrix} 3 \bmod 7 & 0 \bmod 7 & 1 \bmod 7 \\ 1 \bmod 7 & 3 \bmod 7 & 2 \bmod 7 \\ 4 \bmod 7 & 2 \bmod 7 & 2 \bmod 7 \end{pmatrix}$$

Nós checamos isso multiplicando o inverso pela matriz original, a qual rende a matriz identidade sobre o anel coeficiente:

```
>> %*C
```

$$\begin{pmatrix} 1 \bmod 7 & 0 \bmod 7 & 0 \bmod 7 \\ 0 \bmod 7 & 1 \bmod 7 & 0 \bmod 7 \\ 0 \bmod 7 & 0 \bmod 7 & 1 \bmod 7 \end{pmatrix}$$

Exercício 4.41: Nós computamos sobre o anel coeficiente dos números racionais:

```
>> MQ := Dom::Matrix(Dom::Rational):
```

Nós consideramos matrizes 3×3. Para definir a matriz, nós passamos uma função para o construtor que mapeia os índices para as entradas da matriz correspondente:

```
>> A := MQ(3, 3, (i, j) -> (if i=j then 0 else 1 end_if))
```

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

O determinante de A é

```
>> linalg::det(A)
```

$$2$$

As raízes características são as raízes do polinômio característico:

```
>> p := linalg::charpoly(A, x)
```
$$x^3 - 3\,x - 2$$
```
>> solve(p, x)
```
$$\{-1, 2\}$$

Alternativamente, o pacote **linalg** prov~e uma função para computar as raízes características:

```
>> linalg::eigenvalues(A)
```
$$\{-1, 2\}$$

Deixe *I d* denotar a matriz identidade 3 × 3. O espaço característico para a raiz característica $\lambda \in \{-1, 2\}$ é o espaço solução do sistema de equações lineares $(A - \lambda.\ Id) \overrightarrow{x} = \overrightarrow{0}$. Os vetores solução amplitude o espaço nulo (o "kernel") da matriz $A - \lambda \bullet Id$. A função **linalg::nullspace** computa a base para o kernel de uma matriz:

```
>> Id := MQ::identity(3):
>> lambda := -1: linalg::nullspace(A - lambda*Id)
```
$$\left[\begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\right]$$

Há dois vetores base linearmente independentes. Conseqüentemente o espaço característico para a raiz característica $\lambda = -1$ é bidimensional. A outra raiz característica é simples:

```
>> lambda := 2: linalg::nullspace(A - lambda*Id)
```
$$\left[\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\right]$$

Alternativamente, **linalg::eigenvectors** computa todos os espaços característicos simultaneamente:

```
>> linalg::eigenvectors(A)
```
$$\left[\left[-1, 2, \left[\begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}\right]\right], \left[2, 1, \left[\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}\right]\right]\right]$$

O valor retornado é a lista abrigada. Para cada raiz característica λ, ela contém uma lista da forma
[λ, multiplicidade de λ, espaço característico base].

Exercício 4.42:

```
>> p := poly(x^7 - x^4 + x^3 - 1): q := poly(x^3 - 1):
>> p - q^2
```
$$\mathrm{poly}\left(x^7 - x^6 - x^4 + 3\,x^3 - 2, [x]\right)$$

O polinômio *p* é um múltiplo de *q*:

```
>> p/q
```
$$\mathrm{poly}\left(x^4 + 1, [x]\right)$$

Isso é confirmado por uma fatoração:

```
>> factor(p)
```
$$\mathrm{poly}\left(x - 1, [x]\right) \cdot \mathrm{poly}\left(x^2 + x + 1, [x]\right) \cdot \mathrm{poly}\left(x^4 + 1, [x]\right)$$
```
>> factor(q)
```
$$\mathrm{poly}\left(x - 1, [x]\right) \cdot \mathrm{poly}\left(x^2 + x + 1, [x]\right)$$

Exercício 4.43: Nós usamos o identificador **R** para abreviar o tipo do comprimento do nome **Dom::IntegerMod(3)**. Com **alias**, MuPAD também usa **R** como um apelido na saída dos seguintes polinômios:

```
>> p := 3: alias(R = Dom::IntegerMod(p)):
```

Nós apenas precisamos tentar os restos possíveis 0, 1, 2 modulo 3 para os coeficientes *a*, *b*, *c* em *a x*2 + *b x* + *c*. Nós geramos uma lista com todos os 18 polinômios quadráticos com $a \neq 0$ como se segue:

```
>> [((poly(a*x^2 + b*x + c, [x], R) $ a = 1..p-1)
      $ b = 0..p-1) $ c = 0..p-1]:
```

O comando **select(., irreductible)** extrai os 6 polinômios irreduzíveis:

```
>> select(%, irreducible)
          2                          2
   [poly(x  + 1, [x], R), poly(2 x  + x + 1, [x], R),

             2
      poly(2 x  + 2 x + 1, [x], R),

             2                        2
      poly(2 x  + 2, [x], R), poly(x  + x + 2, [x], R),

           2
      poly(x  + 2 x + 2, [x], R)]
```

Exercício 5.1: O valor de **x** é mo identificador **a1**. A avaliação de **x** rende o identificador **c1**. O valor de **y** é o identificador **b2**. A avaliação de **y** rende o identificador **c2**. A avaliação de **z** é o identificador **a3**. A avaliação de **z** tende 10.

A avaliação de **u1** leva para uma recursão infinita, a qual MuPAD aborta com uma mensagem de erro. A avaliação de **u2** rende a expressão **v2^2-1**.

Exercício 6.1: O resultado de **subsop(b + a, 1 = c)** é **b + c** e não **c + a**, como você pode ter esperado. O sistema reordena a soma internamente quando avalia-a, e assim **subsop** processa **a + b** ao invés de **b + a**. Em retorno, o resultado **c + b** é reordenado agora.

Exercício 6.2: A derivada mais alta ocorrendo em **g** é a 6ª derivada **diff(f(x), x$6)**. Nós passamos a seqüência das equações substituídas:

```
diff(f(x), x $ 6) = f6, diff(f(x), x $ 5) = f5, ...,
           diff(f(x), x) = f1, f(x) = f0
```

Para funções de substituição do MuPAD. Note que, de acordo com a notação matemática usual, **diff** retorna a função por si mesma como a 0º derivada: **diff(f(x), x $ 0) = diff(f(x))= f(x).**

```
>> delete f: g := diff(f(x)/diff(f(x), x), x $ 5):
>> subs(g, (diff(f(x), x $ 6 - i) = f.(6-i)) $ i = 0..6)
      4              2
  60 f2    4 f5   20 f3    f0 f6   25 f2 f4
  ------ - ---- + ------ - ----- + -------- -
     4     f1        2       2         2
   f1              f1      f1        f1

            5         2
    120 f0 f2    100 f2  f3   10 f0 f2 f5
    ---------- - ---------- + ----------- +
          6            3           3
         f1           f1          f1

                              2           3
    20 f0 f3 f4   90 f0 f2 f3    240 f0 f2  f3
    ---------- - ----------- + ------------ -
          3             4            5
         f1            f1           f1

           2
    60 f0 f2  f4
    -----------
          4
         f1
```

Exercício 7.1: O seguinte comando rende a avaliação desejada da função:

```
>> f := sin(x)/x: x := 1.23: f
   0.7662510585
```

Contudo, **x** agora tem uma valor. A seguinte chamada **diff(f, x)** vai levar internamente ao comando **diff(0.7662510584, 1.23)** desde que **diff** avalia seus argumentos. Você pode driblar este problema prevenindo uma avaliação completa dos argumentos via **level** ou **hold** (Seção 5.2):

```
>> g := diff(level(f, 1), hold(x)); g
     cos(x)   sin(x)
     ------ - ------
       x        2
               x

   -0.3512303507
```

Aqui a avaliação de **hold(x)** é o identificador **x** e não seu valor. Escrevendo **hold(f)** ao invés de **level (f, 1)** vai render o resultado errado **diff(hold(f), hold(x)) =** 0 desde que **hold(f)** não faça conter **hold(x)**. Usando **level** (f, 1) substitui **f** por seu valor **sin(x)/x** (Seção 5.2). A próxima chamada **g** retorna a avaliação de **g**, nomeada como o valor da derivada em **x = 1.23**. Alternativamente você pode apagar o valor de **x**:

```
>> delete x: diff(f, x): subs(%, x = 1.23); eval(%)
```
$$0.8130081301\cos(1.23) - 0.6609822196\sin(1.23)$$
$$-0.3512303507$$

Exercício 7.2: As primeiras três derivadas do numerador e o denominador desaparecem no ponto $x = 0$:

```
>> Z := x -> (x^3*sin(x)): N := x -> ((1 - cos(x))^2):
>> Z(0), N(0), Z'(0), N'(0), Z''(0), N''(0),
   Z'''(0), N'''(0)
    0, 0, 0, 0, 0, 0, 0, 0
```

Para as quartas derivadas, nós temos:

```
>> Z''''(0), N''''(0)
    24, 6
```

Assim o limite de $Z''''(0)/N''''(0) = 4$, de acordo com a regra de l' Hospital. A função **limit** computa o mesmo resultado:

```
>> limit(Z(x)/N(x), x = 0)
    4
```

Exercício 7.3: A primeira ordem parcial de derivadas de f_1 são:

```
>> f1 := sin(x1*x2): diff(f1, x1), diff(f1, x2)
```
$$x2\cos(x1\,x2),\ x1\cos(x1\,x2)$$

A segunda ordem das derivadas é:

```
>> diff(f1, x1, x1), diff(f1, x1, x2),
   diff(f1, x2, x1), diff(f1, x2, x2)
       2
   - x2  sin(x1 x2), cos(x1 x2) - x1 x2 sin(x1 x2),

                                              2
     cos(x1 x2) - x1 x2 sin(x1 x2), - x1  sin(x1 x2)
```

As derivadas totais de f_2 com respeito a t é:

```
>> f2 := x^2*y^2: x := sin(t): y := cos(t): diff(f2, t)
```
$$2\cos(t)^3\sin(t) - 2\cos(t)\sin(t)^3$$

Exercício 7.4:

```
>> int(sin(x)*cos(x), x = 0..PI/2),
   int(1/(sqrt(1 - x^2)), x = 0..1),
   int(x*arctan(x), x = 0..1),
   int(1/x, x = -2..-1)
```
$$\frac{1}{2},\ \frac{\pi}{2},\ \frac{\pi}{4} - \frac{1}{2},\ -\ln(2)$$

Exercício 7.5:

```
>> int(x/(2*a*x - x^2)^(3/2), x)
```
$$\frac{x}{a\sqrt{2\,a\,x - x^2}}$$
```
>> int(sqrt(x^2 - a^2), x)
```
$$\frac{x\sqrt{x^2 - a^2}}{2} - \frac{a^2\ln\left(x + \sqrt{x^2 - a^2}\right)}{2}$$
```
>> int(1/(x*sqrt(1 + x^2)), x)
```
$$-\operatorname{arctanh}\left(\frac{1}{\sqrt{x^2+1}}\right)$$

Exercício 7.6: A função **intlib::changervar** apenas executa uma mudança de variáveis sem estar convocando a integração:

```
>> intlib::changevar(hold(int)(sin(x)*sqrt(1 + sin(x)),
                     x = -PI/2..PI/2), t = sin(x))
```
$$\int_{-1}^{1} \frac{t\sqrt{t+1}}{\sqrt{1-t^2}}\,dt$$

Uma avaliação futura ativa a rotina de integração:

```
>> eval(%): % = float(%)
```

$$\frac{2\sqrt{2}}{3} = 0.9428090416$$

Quadratura numérica retorna o mesmo resultado:

```
>> numeric::int(sin(x)*sqrt(1 + sin(x)),
                x = -PI/2..PI/2)
```

$$0.9428090416$$

Exercício 8.1: O solucionador de equação retorna a solução geral:

```
>> equations := {a +   b +   c +   d +   e = 1,
                 a + 2*b + 3*c + 4*d + 5*e = 2,
                 a - 2*b - 3*c - 4*d - 5*e = 2,
                 a -   b -   c -   d -   e = 3}:
>> solve(equations, {a, b, c, d, e})
```

$$\{[a = 2, b = d + 2\,e - 3, c = 2 - 3\,e - 2\,d]\}$$

Os parâmetros livres estão nos lados do sentido direito das equações resolvidas. Voc~e pode determinar elas no MuPAD extraindo os lados do sentido direito e usando **indets** para encontrar os identificadores contidos nisto:

```
>> map(%, map, op, 2); indets(%)
```

$$\{[2, d + 2\,e - 3, 2 - 3\,e - 2\,d]\}$$

$$\{d, e\}$$

Exercício 8.2: A solução simbólica é:

```
>> solution := solve(ode(
     {y'(x) = y(x) + 2*z(x), z'(x) = y(x)}, {y(x), z(x)}))
```

$$\left\{\left[z\left(x\right) = \frac{\mathrm{C2}\,\mathrm{e}^{2\,x}}{2} - \mathrm{C1}\,\mathrm{e}^{-x}, y\left(x\right) = \mathrm{C1}\,\mathrm{e}^{-x} + \mathrm{C2}\,\mathrm{e}^{2\,x}\right]\right\}$$

com as constantes livres C1, C2. Nós removemos as chaves exteriores enroladas via **op:**

```
>> solution := op(solution)
```

$$\left[z\left(x\right) = \frac{\mathrm{C2}\,\mathrm{e}^{2\,x}}{2} - \mathrm{C1}\,\mathrm{e}^{-x}, y\left(x\right) = \mathrm{C1}\,\mathrm{e}^{-x} + \mathrm{C2}\,\mathrm{e}^{2\,x}\right]$$

Agora, nós colocamos em série $x = 0$ e substituímos **y(0)** e **z(0)**, respectivamente, para as condições iniciais. Então, nós resolvemos o sistema linear resultante da equação para C1 e C2:

```
>> solve(subs(solution, x = 0, y(0) = 1, z(0) = 1),
        {C1, C2})
```

$$\left\{\left[\mathrm{C1} = -\frac{1}{3}, \mathrm{C2} = \frac{4}{3}\right]\right\}$$

Novamente, nós removemos as chaves onduladas exteriores via **op** e assinamos os valores para C1 e C2 pelos sentidos de **assign**:

```
>> assign(op(%)):
```

Assim, o valor em $x = 1$ da solução simbólica para a condição inicial acima é:

```
>> x := 1: solution
```

$$\left[z\left(1\right) = \frac{\mathrm{e}^{-1}}{3} + \frac{2\,\mathrm{e}^{2}}{3}, y\left(1\right) = \frac{4\,\mathrm{e}^{2}}{3} - \frac{\mathrm{e}^{-1}}{3}\right]$$

Finalmente, nós aplicamos **float**:

```
>> float(%)
```

$$[z\left(1\right) = 5.04866388, y\left(1\right) = 9.729448318]$$

Exercício 8.3:

```
1)  >> solve(ode(y'(x)/y(x)^2 = 1/x, y(x)))
```

$$\left\{\frac{1}{C2 - \ln(x)}\right\}$$

```
2a) >> solve(ode({y'(x) - sin(x)*y(x) = 0, D(y)(1)=1}, y(x)))
```

$$\left\{\frac{e^{\cos(1)}}{\sin(1)\, e^{\cos(x)}}\right\}$$

```
2b) >> solve(ode({2*y'(x) + y(x)/x = 0, D(y)(1) = PI}, y(x)))
```

$$\left\{-\frac{2\,\pi}{\sqrt{x}}\right\}$$

```
3) >> solve(ode({diff(x(t),t) = -3*y(t)*z(t),
                 diff(y(t),t) =  3*x(t)*z(t),
                 diff(z(t),t) = -x(t)*y(t)},
                {x(t),y(t),z(t)}))
                   2      1/2
  {[x(t) = (3 z(t)  - C8)   ,

                      2      1/2
    y(t) = - (- 3 z(t)  - C7)   ],

                     2      1/2
    [x(t) = - (3 z(t)  - C8)   ,

                    2      1/2
    y(t) = (- 3 z(t)  - C7)   ],

                     2      1/2
    [x(t) = - (3 z(t)  - C8)   ,

                      2      1/2
    y(t) = - (- 3 z(t)  - C7)   ],

                   2      1/2
    [x(t) = (3 z(t)  - C8)   ,

                    2      1/2
    y(t) = (- 3 z(t)  - C7)   ]}
```

Exercício 8.4: A função **solve** rende diretamente a solução da recorrência:

```
>> solve(rec(F(n) = F(n-1) + F(n-2), F(n),
             {F(0) = 0, F(1) = 1}))
```

$$\left\{\frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^{n}}{5} - \frac{\sqrt{5}\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^{n}}{5}\right\}$$

Exercício 9.1: Você obtém a resposta imediatamente de:

```
>> simplify(cos(x)^2 + sin(x)*cos(x))
```

$$\frac{\cos(2\,x)}{2} + \frac{\sin(2\,x)}{2} + \frac{1}{2}$$

Você consegue o mesmo resultado aplicando **combine** para reescrever produtos das funções trigonométricas como somas:

```
>> combine(cos(x)^2 + sin(x)*cos(x), sincos)
```

$$\frac{\cos(2\,x)}{2} + \frac{\sin(2\,x)}{2} + \frac{1}{2}$$

Exercício 9.2:

```
1> expand(cos(5*x)/(sin(2*x)*cos(x)^2))
```

$$\frac{\cos(x)^2}{2\sin(x)} - 5\sin(x) + \frac{5\sin(x)^3}{2\cos(x)^2}$$

```
2> f := (sin(x)^2 - exp(2*x)) /
        (sin(x)^2 + 2*sin(x)*exp(x) + exp(2*x)):
>> normal(expand(f))
```

$$-\frac{e^{x} - \sin(x)}{e^{x} + \sin(x)}$$

```
3> f := (sin(2*x) - 5*sin(x)*cos(x)) /
        (sin(x)*(1 + tan(x)^2)):
>> combine(normal(expand(f)), sincos)
```

$$-\frac{3\cos(x)}{\tan(x)^2 + 1}$$

```
4> f := sqrt(14 + 3*sqrt(3 +
                    2*sqrt(5 - 12*sqrt(3 - 2*sqrt(2))))):
>> simplify(f, sqrt)
```

$$\sqrt{2} + 3$$

Exercício 9.3: Como um primeiro passo, nós executamos uma normalização:

```
>> int(sqrt(sin(x) + 1), x): normal(diff(%, x))
```

$$\frac{3\cos(x)^2\sin(x) + \cos(x)^2 + 2\sin(x)^3 - 2\sin(x)}{\cos(x)^2\sqrt{\sin(x)+1}}$$

Então nós eliminamos os termos co-seno:

```
>> rewrite(%, sin)
```

$$\frac{2\sin(x) + 3\sin(x)\left(\sin(x)^2 - 1\right) + \sin(x)^2 - 2\sin(x)^3 - 1}{\left(\sin(x)^2 - 1\right)\sqrt{\sin(x)+1}}$$

O passo final de normalização alcança a simplificação desejada:

```
>> normal(%)
```

$$\sqrt{\sin(x)+1}$$

Na outra mão, chamando **Simplify** é muito mais fácil:

```
>> int(sqrt(sin(x) + 1), x): Simplify(diff(%, x))
```

$$\sqrt{\sin(x)+1}$$

Exercício 9.4: O problema obviamente é que a função avaliação não parece cuidadosamente o bastante no termo para analisar. Para remediar isso, nós usamos a função **lenght** a qual retorna um geral, rapidamente computado "complexidade:"

```
>> noTangent := x -> if has(x, hold(tan))
                    then 1000000
                    else length(x) end_if:
   Simplify(tan(x) - cot(x), Valuation = noTangent)
```

$$-\frac{\cos(2x)}{\cos(x)\sin(x)}$$

Quando nós empregamos **Simplify::defaultValuation** ao invés de **lenght**, MuPAD retorna uma expressão levemente diferente, porque essa função avaliação considera múltiplas ocorrências do mesmo termo (cos (x)) como preferencialmente terminado uns diferentes (cos(x) , cos ($2x$)):

```
>> noTangent := x -> if has(x, hold(tan))
                    then 1000000
                    else Simplify::defaultValuation(x)
                    end_if:
   Simplify(tan(x) - cot(x), Valuation = noTangent)
```

$$-\frac{2\cos(x)^2 - 1}{\cos(x)\sin(x)}$$

Exercício 9.5: A função **assume** (Seção 9.3) determina propriedades para identificadores. Estes são levados dentro em conta por **limit**:

```
>> assume(a > 0): limit(x^a, x = infinity)
```

$$\infty$$

```
>> assume(a = 0): limit(x^a, x = infinity)
```

$$1$$

```
>> assume(a < 0): limit(x^a, x = infinity)
```

$$0$$

Exercício 10.1: Em analogia ao exemplo anterior de maior divisor comum, nós obtemos o seguinte experimento:

```
>> die := random(1..6):
>> experiment := [[die(), die(), die()] $ i = 1..216]:
>> diceScores := map(experiment,
                     x -> (x[1] + x[2] + x[3])):
>> frequencies := Dom::Multiset(op(diceScores)):
>> sortingOrder := (x, y) -> (x[1] < y[1]):
>> sort([op(frequencies)], sortingOrder)
   [[4, 4], [5, 9], [6, 8], [7, 9], [8, 16], [9, 20],

      [10, 27], [11, 31], [12, 32], [13, 20], [14, 13],

      [15, 12], [16, 6], [17, 7], [18, 2]]
```

Neste experimento, a pontuação 3 não ocorre.

Exercício 10.2: a) O comando

```
>> r := float@random(0..10^10)/10^10:
```

Cria um gerador para pontos flutuantes numéricos aleatórios no intervalo [0, 1]. Alternativamente, um pode também usar o (mais rápido) gerador **frandom**. A chamada

```
>> n := 1000: absValues := [sqrt(r()^2+r()^2) $ i = 1..n]:
```

Retorna uma lista com os valores absolutos de n vetores aleatórios no retângulo $Q = [0,1] \times [0,1]$. O número de valores ≤ 1 é o número de pontos aleatórios no quadrante superior direito da unidade do círculo:

```
>> m := nops(select(absValues, z -> (z <= 1)))
    787
```

Desde que **m/n** aproxime a área $\pi/4$ no quadrante superior direito da unidade do círculo, nós obtemos a seguinte aproximação para π:

```
>> float(4*m/n)
    3.148
```

b) Primeiro nós determinamos o máximo de f. A seguinte função plotada mostra que f é de forma monótona acrescentado no intervalo [0, 1]:

```
>> f := x -> x*sin(x) + cos(x)*exp(x):
   plotfunc2d(f(x), x = 0..1):
```

Assim $f(x)$ assume seu valor máximo no fim direito do intervalo. Então, $M = f(1)$ é um limite mais acima para a função: Nós usamos o gerador de número aleatório definido acima para gerar pontos aleatórios no retângulo $[0,1] \times [0, M]$:

```
>> M := f(1.0)
    2.310164925
```

```
>> n := 1000: pointlist := [[r(), M*r()] $ i = 1..n]:
```

Nós selecionamos estes pontos $p = [x, y]$ para cada $0 \leq y \leq f(x)$ mantém:

```
>> select(pointlist, p -> (p[2] <= f(p[1]))):
>> m := nops(%)
    740
```

Assim o seguinte é uma aproximação da integral:

```
>> m/n*M
    1.709522044
```

O valor exato é:

```
>> float(int(f(x), x = 0..1))
    1.679193292
```

Exercício 14.1: Com a seguinte definição do método **postOutput** de **Pref**, o sistema imprime uma linha de status adicional:

```
>> Pref::postOutput(
     proc()
     begin
       "bytes: " .
       expr2text(op(bytes(), 1)) . " (logical) / " .
       expr2text(op(bytes(), 2)) . " (physical)"
      end_proc):
>> DIGITS := 10: float(sum(1/i!, i = 0..100))
    2.718281828

  bytes: 836918 (logical) / 1005938 (physical)
```

Exercício 15.1: Nós primeiro geramos o grupo S:

```
>> f := i -> ( (i^(5/2)+i^2-i^(1/2)-1) /
               (i^(5/2)+i^2+2*i^(3/2)+2*i+i^(1/2)+1)
             ):
>> S := {f(i) $ i=-1000..-2} union {f(i) $ i=0..1000}:
```

Então nós aplicamos **domtype** para todos os elementos do grupo para determinar seus tipos de domínio:

```
>> map(S, domtype)
    {DOM_RAT, DOM_INT, DOM_EXPR}
```

Você vê a explicação para esse resultado olhando em alguns elementos:

```
>> f(-2), f(0), f(1), f(2), f(3), f(4)
```

$$\frac{3i\sqrt{2}+3}{i\sqrt{2}+1},\ -1,\ 0,\ \frac{3\sqrt{2}+3}{9\sqrt{2}+9},\ \frac{8\sqrt{3}+8}{16\sqrt{3}+16},\ \frac{3}{5}$$

A função **normal** simplifica a expressão contendo raízes quadradas:

```
>> map(%, normal)
```

$$3,\ -1,\ 0,\ \frac{1}{3},\ \frac{1}{2},\ \frac{3}{5}$$

Agora nós aplicamos **normal** para todos os elementos do grupo antes examinando seu tipo de dado:

```
>> map(S, domtype@normal)
```

$$\{\text{DOM_RAT}, \text{DOM_INT}\}$$

Assim, todos os números em *S* são realmente racionais (em particular há dois valores integrais **f(0) = -1** e **f(1) = 0**). A razão é que **f(i)** pode ser simplificada para (**i-1**)/(**i+1**):

```
>> normal(f(i) - (i - 1)/(i + 1))
```

$$0$$

Exercício 15.2: Nós aplicamos **testtype(., "sin")** para cada elemento da lista:

```
>> list := [sin(i*PI/200) $ i = 0..100]:
```

para descobrir se isto é retornado na forma **sin**(•) . O seguinte comando **split** (Seção 4.7) decompõe a lista conforme for:

```
>> decomposition := split(list, testtype, "sin"):
```

MuPAD simplifica 9 das 101 chamadas:

```
>> map(decomposition, nops); decomposition[2]
                        [92, 9, 0]

--     1/2              1/2 1/2   1/2       1/2 1/2
|     5           (2 - 2   )     2    (5 - 5   )
|  0, ---- - 1/4, -------------, ------------------,
--     4               2                 4

    1/2   1/2          1/2     1/2
   2     5           (2    + 2)
   ----, ---- + 1/4, -------------,
     2     4               2

    1/2   1/2     1/2    --
   2    (5    + 5)       |
   -----------------, 1  |
            4            --
```

Exercício 15.3: Você pode usar **select** (Seção 4.7) para extrair esses elementos que **testtype** identifica como integrais positivas. Por exemplo:

```
>> set := {-5, 2.3, 2, x, 1/3, 4}:
>> select(set, testtype, Type::PosInt)
```

$$\{2, 4\}$$

Note que isso seleciona apenas estes objetos que *são* integrais positivas, mas não estes que podem *representar* integrais positivas, tais como o identificador **x** no exemplo acima. Isso não é possível com **testtype**. Ao invés disso, você pode usar **assume** para fixar esta propriedade e perguntar isso via **is**:

```
>> assume(x, Type::PosInt):
>> select(set, is, Type::PosInt)
```

$$\{2, 4, x\}$$

Exercício 15.4: Nós construímos o tipo desejado que específica e emprega-o como segue:

```
>> T := Type::ListOf(Type::ListOf(
       Type::AnyType, 3, 3), 2, 2)
```

$$\text{Type::ListOf}\,(\text{Type::ListOf}\,(\text{Type::AnyType}, 3, 3), 2, 2)$$

```
>> testtype([[a, b, c], [1, 2, 3]], T),
   testtype([[a, b, c], [1, 2]], T)
```

$$\text{TRUE},\ \text{FALSE}$$

Exercício 17.1: Considere as condições **x<>1 e A e x=1** ou **A**, respectivamente. Depois entrando:

```
>> x := 1:
```

Não é possível avaliar eles devido a singularidade em $x/(x-1)$:

```
>> x <> 1 and A
   Error: Division by zero [_power]

>> x = 1 or A
   Error: Division by zero [_power]
```

Contudo, isso não é um problema dentro de uma declaração **IF** desde que a avaliação Boolean de **x <> 1n** e **x = 1** já diga-nos que **x <> 1 e A** avalie para **FALSE** e **x=1 ou A** para **TRUE**, respectivamente:

```
>> (if x <> 1 and A then right else wrong end_if),
   (if x = 1 or A then right else wrong end_if)
    wrong, right
```

Na outra mão, avaliando a seguinte declaração **IF** ainda produz um erro, desde que isso é necessário para avaliar **A** na ordem para determinar o verdadeiro valor de **x=1 e A**:

```
>> if x = 1 and A then right else wrong end_if
   Error: Division by zero [_power]
```

Exercício 18.1: Com a fórmula dada, implementação de **"float"** designa é direto:

```
>> ellipticE::float := z -> float(PI/2) *
     hypergeom::float([-1/2, 1/2], [1], z):
   ellipticK::float := z -> float(PI/2) *
     hypergeom::float([1/2, 1/2], [1], z):
```

Isso é quase suficiente:

```
>> ellipticE(1/3)
    E(1/3)
>> float(%)
    1.430315257
```

Contudo, uma coisa está faltando: Nós gostaremos de ter chamadas com argumentos de ponto flutuante avaliados imediatamente:

```
>> ellipticE(0.1)
    E(0.1)
```

Para este fim, nós devemos estender as definições de **ellipticE** e **ellipticK**:

```
>> ellipticE :=
    proc(x) begin
      if domtype(x) = DOM_FLOAT
        or domtype(x) = DOM_COMPLEX
           and (domtype(Re(x)) = DOM_FLOAT
                or domtype(Im(x)) = DOM_FLOAT)
      then
        return(ellipticE::float(x));
      end_if;
      if x = 0 then PI/2
      elif x = 1 then 1
      else procname(x) end_if
    end_proc:
```

Estende **ellipticK** analogamente, então faz essas novas funções dentro de funções ambientes (a definição acima substituída aquela velha completamente) e adiciona a função encaixa nós usamos para as definições originais e os novos encaixes flutuantes, e você obtém:

```
>> ellipticE(0.1)
    1.530757637
```

Exercício 18.2: O seguinte procedimento avalia a função **Abs**:

```
>> Abs := proc(x)
   begin
      if domtype(x) = DOM_INT or domtype(x) = DOM_RAT
         or domtype(x) = DOM_FLOAT
         then if x >= 0 then x else -x end_if;
         else procname(x);
      end_if
   end_proc:
```

Nós convertemos **Abs** para uma função ambiente e suprimos uma função produzindo o desejado na saída da tela:

```
>> Abs := funcenv(Abs,
                  proc(f) begin
                     "|" . expr2text(op(f)) . "|"
                  end_proc,
                  NIL):
```

Então nós agrupamos o atributo da função para diferenciação:

```
>> Abs::diff := proc(f,x) begin
                   f/op(f)*diff(op(f), x)
                end_proc:
```

Então nós temos o seguinte comportamento:

```
>> Abs(-23.4), Abs(x), Abs(x^2 + y - z)
```

```
>> diff(Abs(x^3), x), diff(abs(x^3), x)
                3 |x^3|       2
                -------, 3 |x|  sign(x)
                   x
```

O atributo **diff** da função do sistema **abs** rende um levemente diferente mas resultado equivalente:

Exercício 18.3: Nós usamos **expr2test** (Seção 4.11) para converter as integrais passadas como argumentos para séries. Então nós combinamos eles, junto com alguns cortes, via o operador concatenação "•":

```
>> date := proc(month, day, year) begin
              print(Unquoted, expr2text(month) . "/" .
                              expr2text(day) . "/" .
                              expr2text(year))
          end_proc:
```

Exercício 18.4: Nós apresentamos uma solução usando um loop **while**. A condição **x mod 2=0** checa se **x** é igual:

```
>> f := proc(x) local i;
   begin
     i := 0;
     userinfo(2, "term " . expr2text(i) . ": " .
                 expr2text(x));
     while x <> 1 do
       if x mod 2 = 0 then x := x/2
       else x := 3*x+1 end_if;
       i := i + 1;
       userinfo(2, "term " . expr2text(i) . ": " .
                   expr2text(x))
     end_while;
     i
   end_proc:
>> f(4), f(1234), f(56789), f(123456789)
    2, 132, 60, 177
```

Se nós fixássemos **setuserinfo(f, 2)** (Seção 14.2), então o comando **userinfo** produz todos os termos da seqüência até o procedimento terminar:

```
>> setuserinfo(f, 2): f(4)
  Info: term 0: 4
  Info: term 1: 2
  Info: term 2: 1

    2
```

Se você não acredita na conjectura $3x + 1$, você deve inserir uma condição parando para o índex **i** para segurar o termino.

Exercício 18.5: Uma implementação recursiva baseada na relação máximo divisor comum (a, b) = máximo divisor comum ($a \bmod b$, b) leva para uma recursão infinita: nós temos $a \in \{0,1, \ldots, b-1\}$, e conseqüentemente

$$(a \bmod b) \bmod b = a \bmod b$$

No próximo passo. Assim a função máximo divisor comum vai sempre chamar por si mesma recursivamente com os mesmos argumentos. Contudo, uma chamada recursiva da forma máximo divisor comum (a, b) = *máximo divisor comum*(b, $a \bmod b$) fazer sentido. Desde que $\square \bmod b < b$, a função chama por si mesma recursivamente pelo decréscimo dos valores do segundo argumento, os quias finalmente se tornam zero:

```
>> Gcd := proc(a, b) begin     /* recursive variant */
            if b = 0
               then a
               else Gcd(b, a mod b)
            end_if
         end_proc:
```

Para grandes valores de a e b, você pode precisar acrescentar o valor da tabela ambiente **MAXDEPTH** de Máximo divisor comum esvazia a recursão válida profundidade. A seguinte variante repetitiva esvazia este problema:

```
>> GCD := proc(a, b) local c;  /* iterative variant */
         begin
            while b <> 0 do
                c := a; a := b; b := c mod b
            end_while;
            a
         end_proc:
```

Essa implementação rende o mesmo resultado como a função **igcd** e **gcd** provida pelo sistema:

```
>> a := 123456: b := 102880:
>> Gcd(a, b), GCD(a, b), igcd(a, b), gcd(a, b)
    20576, 20576, 20576, 20576
```

Exercício 18.6: Na seguinte implementação, nós geramos uma cópia encurtada $Y = [x_1, \ldots, x_n]$ de $X = [x_0, \ldots, x_n]$ e computamos a lista de diferenças $[x_1 - x_0, \ldots, x_n - x_{n-1}]$ via **zip** e **_subtract** (**_subtract(y,x)=y-x)**. Então nós multiplicamos cada elemento dessa lista com o valor numérico correspondente na lista $[f(x_0), f(x_1), \ldots]$. Finalmente, a função **_plus** adiciona todos os elementos da lista resultante:

$$[(x_1 - x_0)f(x_0), \ldots, (x_n - x_{n-1})f(x_{n-1})]:$$

```
>> Quadrature := proc(f, X)
   local Y, distances, numericalValues, products;
   begin
      Y := X; delete Y[1];
      distances := zip(Y, X, _subtract);
      numericalValues := map(X, float@f);
      products := zip(distances, numericalValues, _mult);
      _plus(op(products))
   end_proc:
```

No exemplo seguinte, nós usamos $n = 1000$ pontos eqüidistantes exemplares no intervalo [0, 1]:

```
>> f := x -> (x*exp(x)): n := 1000:
>> Quadrature(f, [i/n $ i = 0..n])
    0.9986412288
```

Essa é a aproximação numérica (crua) de $\int_0^1 x\ ex\ dx$ (= 1).

Exercício 18.7: A especificação de **Newton** requer que o primeiro argumento **f** seja uma *expressão* e não uma função do MuPAD. Assim, para computar a derivada, nós usamos primeiro **indets** para determinar a incógnita em **f**. Nós substituímos o valor numérico para a incógnita para avaliar a repetição da função $F(x) = x - f(x)/f'(x)$ num ponto:

```
>> Newton := proc(f, x0, n)
     local vars, x, F, sequence, i;
     begin
       vars := indets(float(f)):
       if nops(vars) <> 1
          then error(
        "the function must contain exactly one unknown"
                    )
          else x := op(vars)
       end_if;
       F := x - f/diff(f,x); sequence := x0;
       for i from 1 to n do
           x0 := float(subs(F, x = x0));
           sequence := sequence, x0
       end_for;
       return(sequence)
     end_proc:
```

No seguinte exemplo, **Newton** computa os primeiros termos de uma seqüência rapidamente convergindo para a solução $\sqrt{2}$:

```
>> Newton(x^2 - 2, 1, 6)
   1, 1.5, 1.416666667, 1.414215686, 1.414213562,

      1.414213562, 1.414213562
```

Exercício 18.8: A chamada **numlib::g_adic**(...,2) rende a expansão binária de uma integral como uma lista de bits:

```
>> numlib::g_adic(7, 2), numlib::g_adic(16, 2)
    [1, 1, 1], [0, 0, 0, 0, 1]
```

Ao invés de chamando **numlib::g_adic** diretamente, nossa solução usa um sub-procedimento **binary** fornecido com a função **option remember**. Isso acelera a computação notavelmente desde que **numlib::g_adic** é chamado freqüentemente com os mesmos argumentos. A chamada **isPoint([x, y])** retorna **TRUE** quando o ponto especificado pela lista **[x, y]** é um ponto de Sierpinski. Para checar isso, a função multiplica as listas com os bits das duas coordenadas.

Nestas posições onde ambos **x** e **y** tem um 1 bit, multiplicação rende um 1. Em todos os outros casos, $0 \cdot 0$, $1 \cdot 0$, $0 \cdot 1$, produz o resultado 0. Se a lista dos produtos contém ao menos um 1, o ponto é um ponto Sierpinski. Nós usamos **select** (Seção 4.6) para extrair os pontos Sierpinski de todos os pontos considerados. Finalmente, nós criamos um **plot::PointList2d** com esses pontos e chama **plot** (Seção 11):

```
>> Sierpinski := proc(xmax, ymax)
   local binary, isSPoint, allPoints, i, j, SPoints;
   begin
     binary := proc(x) option remember; begin
                 numlib::g_adic(x, 2)
               end_proc;
     isSPoint := proc(Point) local x, y; begin
                   x := binary(Point[1]);
                   y := binary(Point[2]);
                   has(zip(x, y, _mult), 1)
                 end_proc;
     allPoints := [([i, j] $ i = 1..xmax) $ j = 1..ymax];
     SPoints := select(allPoints, isSPoint);
     plot(plot::PointList2d(SPoints, Color = RGB::Black));
   end_proc:
```

Para **xmax** = **ymax** = **100**, dizer, você obtém uma foto completamente interessante:

```
>> Sierpinski(100, 100)
```

Exercício 18.9: Nós apresentamos uma solução recursiva. Dada uma expressão **formula(x1, x2,...)** com os identificadores **x1, x2,...,** nós coletamos os identificadores no jogo **x={x1, x2,...}** pelos sentidos de **indets**. Então nós substituímos **TRUE** e **FALSE**, respectivamente, por **x1 (=op(x, 1))**, e chamamos o procedimento recursivamente com os argumentos **formula (TRUE, x2, x3,...)** e **formula (FALSE, X2, X3, ...)**, respectivamente. Deste modo, nós testamos todas as combinações possíveis de **TRUE** e **FALSE** para os identificadores até a expressão poder finalmente ser simplificada para **TRUE** ou **FALSE** na base da recursão.
Esse valor é retornado para um procedimento chamando.ç Se ao menos uma das combinações **TRUE**/**FALSE** rende **TRUE**, então o procedimento retorna **TRUE**, indicando que a fórmula é satisfatória, e do contrário ela retorna **FALSE**.
Se o número de identificadores na entrada da fórmula é *n*, então a profundidade da recursão é na maioria *n* e o número total de chamadas recursivas do procedimento é na maioria 2^n. Nós aplicamos este procedimento em dois exemplos:

```
>> F1 := ((x and y) or (y or z)) and (not x) and y and z:
>> F2 := ((x and y) or (y or z)) and (not y) and (not z):
>> satisfiable(F1), satisfiable(not F1),
   satisfiable(F2), satisfiable(not F2)
    TRUE, TRUE, FALSE, TRUE
```

A chamada **simplify(•, logic)** (Seção 9.2) simplifica formulário lógico. Formula **F2** pode ser simplificada para false, não importando o que o valor de *x*, *y*, e *z* são:

```
>> simplify(F1, logic), simplify(F2, logic)
    ¬x ∧ y ∧ z, FALSE
```